Pool-uri de fire de execuție în Java

Șablonul Thread Pool ajută la economisirea resurselor într-o aplicație multithreaded și la conținerea paralelismului în anumite limite predefinite.

Când folosim un pool de thread-uri, **scriem codul nostru concurent sub formă de sarcini paralele și le trimitem spre execuție către o instanță a unui pool de thread-uri.** Această instanță controlează mai multe thread-uri reutilizate pentru executarea acestor sarcini.

1. **Executor Framework**

- ***Interfața Executor*** oferă o modalitate standard de a trimite sarcini pentru execuție. Aceasta decuplează trimiterea sarcinilor de execuția sarcinilor, permițând o gestionare flexibilă a firelor de execuție.
- ***Interfața ExecutorService*** extinde Executor și adaugă metode pentru gestionarea ciclului de viață al executorului, cum ar fi închiderea pool-ului și așteptarea terminării.
- ***Clasa de utilitare Executors*** oferă metode factory pentru crearea diferitelor tipuri de instanțe ExecutorService, inclusiv pool-uri de thread-uri fixe, pool-uri de thread-uri memorate în cache și executori cu un singur fir de execuție.

Exemplu de utilizare a ExecutorService:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);

// Creează un pool de thread-uri cu 5 thread-uri

        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            executor.submit(() -> {

// Trimite o sarcină (Runnable) către executor
                System.out.println("Executing task " + taskId + " in thread: "
+ Thread.currentThread().getName());
            });
        }

        executor.shutdown(); // Inițiază o oprire ordonată
    }
}
```

2. **Thread Pool (ThreadPoolExecutor)**

• ***Un thread pool este o colecție de thread-uri worker*** care pot executa mai multe sarcini. Reutilizează thread-urile existente, reducând costurile suplimentare de creare și distrugere a thread-urilor pentru fiecare sarcină.

• ***ThreadPoolExecutor este clasa*** principală pentru crearea și gestionarea pool-urilor de thread-uri personalizate. Permite un control detaliat asupra parametrilor precum dimensiunea pool-ului de bază, dimensiunea maximă a pool-ului, timpul de menținere a activității și tipul cozii.

Exemplu de ThreadPoolExecutor:

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExample {
    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                2, // Dimensiunea CorePool
                5, // Dimensiune maximă a pool
                60, TimeUnit.SECONDS, // timpul de menținere în viață
                new ArrayBlockingQueue<>(10) // Coadă de lucru
        );

        for (int i = 0; i < 15; i++) {
            final int taskId = i;
            executor.submit(() -> {
                System.out.println("Executing task " + taskId + " in thread: "
+ Thread.currentThread().getName());
                try {
                    Thread.sleep(100); // Simulate work
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        executor.shutdown();
    }
}
```

3. Framework-ul Fork/Join

• Framework-ul Fork/Join este o implementare a ExecutorService optimizată pentru sarcini care pot fi împărțite recursiv în subsarcini mai mici (împărțire și constrângere).

• Folosește un algoritm de furt de lucru, în care firele de execuție inactive pot „fura” sarcini din cozile firelor de execuție ocupate, asigurând o utilizare eficientă a resurselor. • Clasele cheie sunt ForkJoinPool, RecursiveTask (pentru sarcinile care returnează un rezultat) și RecursiveAction (pentru sarcinile care nu returnează un rezultat).

Exemplu de framework Fork/Join:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class ForkJoinExample {
    static class SumTask extends RecursiveTask<Long> {
        private final long[] array;
        private final int start;
        private final int end;
        private static final int THRESHOLD = 1000;

        public SumTask(long[] array, int start, int end) {
            this.array = array;
            this.start = start;
            this.end = end;
        }

        @Override
        protected Long compute() {
```

```
            if (end - start <= THRESHOLD) {
                long sum = 0;
                for (int i = start; i < end; i++) {
                    sum += array[i];
                }
                return sum;
            } else {
                int mid = start + (end - start) / 2;
                SumTask leftTask = new SumTask(array, start, mid);
                SumTask rightTask = new SumTask(array, mid, end);

                leftTask.fork(); // Execută asincron subtasking-ul din stânga
                long rightResult = rightTask.compute();

// Execută sincron subtasking-ul corect
                long leftResult = leftTask.join();

// Așteptați și recuperați rezultatul subsarcinii din stânga
                return leftResult + rightResult;
            }
        }
    }

    public static void main(String[] args) {
        long[] array = new long[100000];
        for (int i = 0; i < array.length; i++) {
            array[i] = i + 1;
        }

        ForkJoinPool pool = new ForkJoinPool();
        long sum = pool.invoke(new SumTask(array, 0, array.length));
        System.out.println("Total sum: " + sum);
        pool.shutdown();
    }
}
```

Lucrare de laborator nr. 4

Tema lucrării: ***Pool-uri de fire de execuție în Java. Sincronizarea thread-urilor în Java.***

Scopul lucrării:

Este studierea și aplicarea mecanismelor de gestionare și sincronizare a firelor de execuție în Java, prin utilizarea **pool-urilor de thread-uri** și **a conceptului clasic producător–consumator**, pentru a asigura execuția concurentă eficientă și sigură a proceselor.

Obiectivele lucrării:

4. **Înțelegerea conceptului de thread pooling** și a avantajelor utilizării clasei `ExecutorService` față de crearea manuală a firelor.
5. **Implementarea modelului producător–consumator** folosind mecanisme de sincronizare (`wait()`, `notify()`, `synchronized`, `BlockingQueue`).
6. **Aplicarea clasei `Executors`** pentru a crea un pool fix de fire (`FixedThreadPool`).
7. **Analiza comportamentului concurent** și a problemelor de sincronizare (blocare, interferență, acces simultan).
8. **Dezvoltarea unei aplicații Java practice** în care mai mulți producători și consumatori partajează un depozit comun.

9. **Măsurarea performanței** la diferite dimensiuni ale pool-ului de thread-uri și a capacității depozitului.

Exemplu de realizare:

```
import java.util.concurrent.*;
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;

public class ProducerConsumerExecutorFixed {

    private static final int BUFFER_CAPACITY = 9;
    private static final BlockingQueue<Character> buffer = new
ArrayBlockingQueue<>(BUFFER_CAPACITY);

    private static final int CONSUMER_GOAL = 13;
    private static final int PRODUCER_COUNT = 3;
    private static final int CONSUMER_COUNT = 4;

    private static final int TOTAL_OBJECTS = CONSUMER_GOAL * CONSUMER_COUNT; // 52

    // Contoare atomice
    private static final AtomicInteger totalProduced = new AtomicInteger(0);
    private static final AtomicInteger totalConsumed = new AtomicInteger(0);
    private static final Map<Integer, AtomicInteger> consumerCounters = new
ConcurrentHashMap<>();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(PRODUCER_COUNT +
CONSUMER_COUNT);

        for (int i = 1; i <= CONSUMER_COUNT; i++) {
            consumerCounters.put(i, new AtomicInteger(0));
        }

        // Pornim producătorii
        for (int i = 1; i <= PRODUCER_COUNT; i++) {
            executor.execute(new Producer(i));
        }

        // Pornim consumatorii
        for (int i = 1; i <= CONSUMER_COUNT; i++) {
            executor.execute(new Consumer(i));
        }

        executor.shutdown();

        try {
            // Așteptăm ca toate thread-urile să se termine
            executor.awaitTermination(60, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("\n========== RAPORT FINAL ==========");
        System.out.println("Total produse: " + totalProduced.get());
        System.out.println("Total consumate: " + totalConsumed.get());
        consumerCounters.forEach((id, count) ->
                System.out.println("Consumator " + id + ": " + count.get() + "
obiecte consumate"));
        System.out.println("==================================");
    }
```

```
    // ================================
    // CLASA PRODUCĂTOR
    // ================================
    static class Producer implements Runnable {
        private final int id;
        private final Random random = new Random();
        private final char[] consonants = "BCDFGHJKLMNPQRSTVWXYZ".toCharArray();

        Producer(int id) {
            this.id = id;
        }

        @Override
        public void run() {
            try {
                while (totalProduced.get() < TOTAL_OBJECTS) {
                    char item = consonants[random.nextInt(consonants.length)];

                    if (buffer.remainingCapacity() == 0) {
                        System.out.println("⚠  [Producător " + id + "] Depozitul e
plin, așteaptă...");
                    }

                    buffer.put(item);
                    int produced = totalProduced.incrementAndGet();

                    System.out.println("🧱 [Producător " + id + "] a produs: " +
item +
                            " | Total produse: " + produced +
                            " | Capacitate curentă: " + buffer.size() + "/" +
BUFFER_CAPACITY);
                 // Thread.sleep(200);

                    if (produced >= TOTAL_OBJECTS) {
                          System.out.println("Producatorul  " + id + " s-a
finalizat");
                          break;
                    }

                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    // ================================
    // CLASA CONSUMATOR
    // ================================
    static class Consumer implements Runnable {
        private final int id;

        Consumer(int id) {
            this.id = id;
        }

        @Override
        public void run() {
            try {
                while (consumerCounters.get(id).get() < CONSUMER_GOAL) {
                    if (buffer.isEmpty()) {
```

```
                    System.out.println("⚠  [Consumator " + id + "] Depozitul e
gol, așteaptă...");
                }

                char item = buffer.take();
                consumerCounters.get(id).incrementAndGet();
                int consumed = totalConsumed.incrementAndGet();

                System.out.println("🍽 [Consumator " + id + "] a consumat: " +
item +
                        " | Total consumate: " + consumerCounters.get(id).get() +
                        " | În depozit: " + buffer.size() +
                        " | Total global: " + consumed);

                Thread.sleep(300);
            }

            System.out.println("✅ [Consumator " + id + "] a fost îndestulat cu
13 obiecte!");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
}
```

Probleme propuse spre realizare:

Sunt dați X producători care generează aleatoriu F obiecte care sunt consumate de Y consumatori. De afişat informaţia despre producerea şi consumarea obiectelor, mesajele despre cazurile când “depozitul e gol sau plin”. Toate operaţiile se efectuează până când fiecare consumator este îndestulat cu Z obiecte.

Dimensiunea depozitului este D. Valorile pentru X, Y, Z, D, F sunt indicate în Tabelul 3.

Tabelul 1 Variantele pentru realizarea sarcinii

Nr.	X	Y	Z	D	Tip Obiecte
1	2	3	11	8	Numere pare
2	3	4	2	5	Numere impare
3	4	3	3	10	Vocale
4	3	2	12	11	Consoane
5	2	5	3	12	Numere pare
6	2	4	4	7	Numere impare
7	3	3	5	6	Vocale
8	4	2	4	5	Consoane
9	5	2	3	4	Numere pare

10	3	3	5	2	Numere impare
F - fiecare producător produce câte 2 obiecte de fiecare dată pentru toate variantele					

Criterii de evaluare:

1. Crearea şi iniţializarea thread-urilor pentru realizarea sarcinilor.
2. Alegerea formelor de sincronizare a firelor.
3. Sincronizarea thread-urilor cu formele potrivite.
4. Crearea interfeţei grafice a programului.
5. **Corectitudinea codului** - verificarea dacă codul este corect, fără erori de funcționare.
6. **Respectarea instrucțiunilor și cerințelor** - verificarea corectitudenii cerințelor sarcinii, cum ar fi numărul de fire de execuție.
7. **Optimizarea codului** - Evaluarea eficienței codului în utilizarea resurselor și evitarea codului.
8. **Respectarea termenului de susținere** - evaluarea punctajului în funcție de punctualitate, dacă lucrarea a fost predată în termenul stabilit.
9. **Evaluarea cunoștințelor** - explicațiile oferite despre procesul de realizare a lucrării, ceea ce poate include descrierea funcțiilor principale și a logicii utilizate.
10. Utilizarea de către student a IA.

Pentru obținerea notei 5 - 6 sunt obligatorii criteriile 1,2,3,5,8,9
Pentru obținerea notei 7 - 8 sunt obligatorii criteriile 1,2,3,4,5,6,8,9
Pentru obținerea notei 9 - 10 sunt obligatorii criteriile 1-9
Dacă a fost utilizat criteriul 10 nota este scăzută cu 2 baluri, numai dacă studentul sa lămurit în funcționarea codului. În caz contrar lucrare de laborator nu este susținută.

Întrebări de verificare

1. Ce este un *thread pool* și care sunt avantajele sale față de crearea manuală a firelor?
2. Ce rol joacă interfața `Executor` și clasa `ExecutorService` în Java?
3. Ce metode principale oferă `ExecutorService` pentru gestionarea firelor?
4. În ce constă problema producător–consumator?
5. Cum se poate implementa această problemă folosind `BlockingQueue`?
6. Ce diferență există între `synchronized`, `ReentrantLock` și mecanismele din `java.util.concurrent`?
7. Ce reprezintă metodele `wait()`, `notify()`, `notifyAll()` și în ce condiții pot fi apelate?
8. Cum se încheie corect execuția unui pool de fire (`shutdown()` vs `shutdownNow()`)?
9. Ce se întâmplă dacă numărul de producători este mai mare decât cel de consumatori?
10. Care sunt principalele probleme de sincronizare care pot apărea într-un sistem concurent?

Lista de literatură recomandată

1. Balaurea, M. – *Programarea concurentă și distribuită în Java*, Editura Universității Tehnice, Chișinău, 2022.
2. Moțoc, I. – *Programare avansată în Java. Fire de execuție și sincronizare*, Editura Polirom, Iași, 2020.
3. Popa, C. – *Concurența și paralelismul în Java*, Editura MatrixRom, București, 2018.

4. Documentația oficială Oracle (secțiunea *Concurență și Executor Framework*): https://docs.oracle.com/javase/tutorial/essential/concurrency/
5. Хорстманн К. – *Java. Библиотека профессионала. Том 2: Расширенные средства программирования*, СПб: Питер, 2021.
6. Шилдт Г. – *Java: Руководство для начинающих*, 12-е издание, Москва: Вильямс, 2022.
7. Головач С., Романчук В. – *Java. Многопоточность. Синхронизация и параллельность*, Киев, 2020.
8. Документация Oracle по многопоточности: https://docs.oracle.com/javase/tutorial/essential/concurrency/
9. Goetz, B., Peierls, T. – *Java Concurrency in Practice*, Addison-Wesley, 2006.
10. Bloch, J. – *Effective Java*, 3rd Edition, Addison-Wesley, 2018.
11. Oracle Docs – *Concurrency Utilities and Executors*: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
12. Lea, D. – *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 2nd Edition, 2000.