Istruzioni

- Tempo a disposizione: 2 ore
- L'esame è composto da domande a risposta multipla, domande teoriche a risposta breve e esercizi pratici
- Per le domande a risposta multipla, selezionare una o più risposte corrette come indicato
- Non è consentito l'uso di materiale didattico

Parte 1: Domande a Risposta Multipla (1 punto ciascuna)

1. Il teorema CAP afferma che un sistema distribuito, in caso di partizione dei suoi nodi in gruppi che non comunicano fra loro, deve scegliere il suo comportamento fra:

A) Consistenza e latenza B) Correttezza ed efficienza C) Consistenza e disponibilità D) Certezza delle risposte e latenza della stessa

2. L'interfaccia Collector permette di eseguire la riduzione ad un risultato di uno Stream parallelo in modo più efficiente perché:

A) Non necessita di sapere la lunghezza dello Stream B) Gestisce un accumulatore mutabile, che riduce la pressione sulla Garbage Collection C) Combina i risultati intermedi più velocemente D) Mantiene il parallelismo dello Stream

3. Come molti altri linguaggi Object-Oriented, Java ha a disposizione un meccanismo di ereditarietà per estendere classi esistenti senza doverle modificare. L'ereditarietà in Java ha le seguenti caratteristiche (selezionare tutte le risposte corrette):

A) A causa dell'introduzione dei default methods, è possibile causare un Diamond Problem B) Una interfaccia può implementare una sola altra interfaccia C) Una classe può implementare più interfacce D) Il Diamond Problem è impossibile per costruzione E) Una classe può ereditare da una sola altra classe F) Una interfaccia può ereditare da un'altra interfaccia

4. La comunicazione su Datagram presenta alcuni vantaggi rispetto ai Socket, ma è afflitta anche dai seguenti svantaggi

(selezionare tutte le risposte corrette):

A) Le due parti devono concordare in qualche modo l'encoding delle stringhe all'interno del protocollo di comunicazione B) Deve essere definito un protocollo con cui i due lati della comunicazione riconoscono inizio e la fine dei messaggi C) La probabilità di successo della comunicazione diminuisce con il crescere della lunghezza del messaggio D) Le due parti della comunicazione devono inviare i propri dati il più velocemente possibile per diminuire la latenza della comunicazione

5. Quali di questi elementi fanno parte della firma di un metodo (selezionare tutte le risposte corrette):

A) Nome del metodo B) Visibilità del metodo C) Tipo del valore di ritorno D) Parametri di tipo E) Elenco dei tipi degli argomenti F) Elenco dei nomi degli argomenti

6. Gli Stream permettono di rendere parallela l'esecuzione della pipeline delle operazioni definite su di essi, tuttavia non permettono di indicare esplicitamente il grado di parallelismo da usare. In quali casi può essere necessario modificare il comportamento di default (selezionare tutte le risposte corrette):

A) Un algoritmo che genera molta I/O può beneficiare dall'essere parallelizzato su di un numero maggiore di Threads rispetto al default B) Un algoritmo che occupa costantemente la CPU può beneficiare dall'essere parallelizzato su di un numero maggiore di Threads rispetto al default C) Un algoritmo che occupa costantemente la CPU può beneficiare dall'essere parallelizzato su di un numero inferiore di Threads rispetto al default D) Un algoritmo che comporta molti cambi di contesto può beneficiare dall'essere parallelizzato su di un numero inferiore di Threads rispetto al default

7. In questo codice di esempio:

```
try (
    ServerSocket serverSocket = new ServerSocket(portNumber);
    Socket socket = serverSocket.accept();
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
) {
    String inputLine;
    while ((inputLine = in.readLine()) != null) {
        System.out.println("Received: " + inputLine);
        out.println("Hello " + inputLine);
```

```
    }
}
```

Una eccezione lanciata dal metodo out.println() ha come effetto: A) La chiusura della sola risorsa in B) L'uscita dal blocco con perdita delle risorse allocate C) La chiusura delle risorse serverSocket e socket, ma non di in D) La chiusura di tutte le risorse e l'uscita dal blocco

8. Associare ad ogni condizione di Coffman una strategia utile per rimuoverla. Una delle strategie indicate non è rilevante:

1. Mutua Esclusione
2. Possesso e attesa
3. Risorse non riassegnabili
4. Attesa circolare
5. Non rilevante

a) Algoritmi lock-free b) Assegnazione delle risorse transazionale c) Pre-emption d) Ordinamento dell'acquisizione e) Esecuzione fuori ordine

9. Quale delle seguenti affermazioni riguardo l'istruzione return è corretta:

A) In un metodo void è necessario inserire almeno una istruzione return vuota B) In un metodo void, se viene indicato un valore questo viene ignorato C) In un metodo non void, ogni percorso di codice deve terminare con una istruzione return D) In un metodo non void è opzionale, in quanto viene ritornato il risultato dell'ultima espressione del metodo

10. Quali dei seguenti possono essere indicati come vantaggi dell'uso dell'oggetto Socket per la comunicazione (selezionare tutte le risposte corrette):

A) I buffer a disposizione per le connessioni su Socket sono più ampi di quelli per i Datagram B) L'orientamento alla connessione del Socket permette di trasferire in modo affidabile quantità di dati rilevanti C) La comunicazione può essere bidirezionale D) La connessione con un Socket non è legata ad una specifica porta del nodo connesso

11. Quale delle seguenti frasi è falsa per una struttura dati non Thread-Safe in caso di accesso concorrente:

A) Può lanciare un'eccezione per segnalare un accesso non consentito o pericoloso B) È meno costosa in termini di cicli di CPU che nel caso di accesso esclusivo C) Può dare un risultato errato o venirsi a trovare in uno stato inconsistente D) È più performante nel caso generale

12. Quando si dice che il compilatore Java ha delle capacità di Type Inference si intende che:

A) È in grado di indicare se il grafo dell'ereditarietà genera un diamond problem B) È in grado di calcolare la corretta indentazione del codice e correggerla C) È in grado di dedurre il tipo di alcune espressioni senza che sia necessario indicarlo esplicitamente D) È in grado di trasformare un tipo in un altro senza indicazioni esterne

13. Nel linguaggio Java, una variabile final:

A) Richiama un metodo quando viene modificata B) Può contenere un valore di un solo tipo C) Deve essere inizializzata contestualmente alla definizione, ed il suo valore non può essere cambiato D) Richiama un metodo al momento della cancellazione

14. La fallacia "The network is homogeneus" è stata aggiunta alle prime sette da Gosling, proprio in seguito alle prime esperienze con Java. Oggi, la sua rilevanza:

A) Non è più rilevante dopo la diffusione delle connessioni wireless B) È ancora rilevante, ma si tratta ormai di un problema ormai sotto controllo C) È tutto sommato un problema risolto dalla diffusione dei protocolli più recenti D) È ancora maggiore, perché le tipologie e le caratteristiche delle reti sono sempre più varie

15. In un sistema che implementa la specifica Reactive Streams, si intende con back-pressure:

A) La latenza introdotta dal nodo più lento dello stream B) La possibilità per un nodo di indicare al precedente quanti oggetti è in grado di elaborare C) La quantità di dati gestibile dalle interfacce di rete fra i nodi D) La possibilità per un nodo di indicare al successivo quanti oggetti è in grado di inviargli

16. Con i metodi di esecuzione nativa, il codice Java viene compilato direttamente in un eseguibile, senza la necessità di usare la JVM. Questo comporta (selezionare tutte le risposte corrette):

A) Prestazioni a regime che possono essere superiori a quelle dell'esecuzione normale (JIT) B) Una diminuzione delle risorse necessarie durante l'esecuzione C) Un tempo di compilazione più lungo D) Un aumento delle risorse necessarie durante l'esecuzione E) Un avvio più rapido dell'applicazione F) Prestazioni a regime che possono essere inferiori a quelle dell'esecuzione normale (JIT)

17. Le classi del package java.concurrent.atomic:

A) Non sono adatte nel caso di modifica concorrente perché permettono una sola modifica alla volta B) Sono particolarmente efficienti in caso di modifica concorrente del dato che rappresentano perché usano (se disponibili) delle funzionalità fornite direttamente dall'hardware C) Non sono adatte nel caso di modifica concorrente perché permettono di leggere un dato che in realtà è già stato modificato D) Sono particolarmente efficienti in caso di modifica concorrente del dato che rappresentano perché usano nel modo migliore i lock

18. Un Thread esce dallo stato running quando (selezionare tutte le risposte corrette):

A) Termina l'esecuzione, passando allo stato terminated B) Gli viene sottratta la CPU, e passa allo stato runnable C) Si pone in attesa di una risorsa, e passa allo stato waiting D) Gli viene assegnata la CPU, e passa allo stato runnable E) Si pone in attesa di una risorsa per un tempo limitato, passando allo stato timed_waiting

19. Un oggetto Future rappresenta:

A) Il risultato di un calcolo parallelo terminato correttamente B) Un calcolo che potrebbe produrre un risultato dopo un certo tempo C) Una generica esecuzione concorrente D) Un calcolo concorrente terminato in modo errato

20. Una variabile di tipo ThreadLocal:

A) Più Thread possono accedere allo stesso valore senza interferire fra loro B) Permette ad più Thread di accedere rapidamente al valore che contiene C) Possiede un valore differente per ogni Thread che vi accede D) Un solo Thread per volta può accedere al valore contenuto

21. Una classe Java dichiarata senza modificatori di visibilità:

A) Un modificatore di visibilità è obbligatorio B) È visibile da ogni classe del sistema C) Da ogni classe dello stesso package D) Solo dalle classe che la estendono

22. Quali delle seguenti affermazioni sono corrette riguardo al comportamento di un Attore (selezionare tutte le risposte corrette):

A) Il comportamento di un attore può cambiare dopo la ricezione di un messaggio B) Il comportamento di un attore deve essere thread-safe C) Il comportamento di un attore non può agire sul suo stato interno D) Il comportamento di un attore definisce come reagisce ad un messaggio E) Il comportamento di un attore non può cambiare dopo la sua creazione

23. È importante gestire velocemente l'accettazione di un pacchetto da una DatagramSocket perché:

A) Se c'è un thread in attesa su DatagramSocket::receive(), gli invii sono bloccati B) Se non c'è un thread in attesa della ricezione di un pacchetto, questo viene scartato C) Se non c'è un thread in attesa su DatagramSocket::receive(), nessun pacchetto viene ricevuto D) I pacchetti ricevuti vengono conservati in buffer limitati; se si riempiono, i messaggi successivi sono scartati

24. Con la parola chiave sealed si può indicare una classe che:

A) Non può essere estesa B) È digitalmente firmata per sicurezza C) Elenca esplicitamente le classi che possono ereditare da essa D) Può essere istanziata solo in oggetti immutabili

25. Uno stream rappresenta una sequenza di elementi, potenzialmente infinita. L'obiettivo di questa astrazione è:

A) Uniformare l'accesso a più collezioni di struttura differente B) Permettere di descrivere l'elaborazione in termini delle operazioni sugli elementi, e non dell'avanzamento dell'iterazione C) Fornire l'API per attraversare più rapidamente la collezione D) Permettere di controllare più finemente l'avanzamento dell'iterazione

26. Un Observable nel modello Reactive Extension si distingue radicalmente da uno Stream della libreria standard di Java perché:

A) Emette elementi nel tempo. Lo scorrere del tempo è un concetto esplicitamente gestito dal modello Rx B) Può trovarsi su più nodi di calcolo e distribuire il carico fra di essi C) È più efficiente nel costruire ed eseguire la pipeline di elaborazione D) Non emette errori. Ogni possibile casistica d'errore è gestita a priori

27. I Virtual Threads vengono introdotti per (selezionare tutte le risposte corrette):

A) Aumentare l'efficienza del codice che usa operazioni bloccanti senza doverlo modificare B) Rendere più efficiente il codice reattivo C) Aumentare il livello di prestazioni che si può ottenere senza dover riscrivere il codice in un altro paradigma, per es. reattivo D) Aumentare le prestazioni dei carichi a basso Blocking Factor

28. Quali vantaggi si cercano nel distribuire lo stato di un sistema su più nodi (selezionare tutte le risposte corrette):

A) Elaborazione più rapida delle richieste distribuite a più nodi B) Maggiore sicurezza dei dati gestiti dal sistema di consenso C) Possibilità di gestire uno stato più grande della capacità di una singola macchina D) Accesso più rapido da località differenti

29. Quando di un sistema reattivo si indicano le sue qualità come Responsive, Resilient, Elastic, Message-Oriented, con la qualità "Elastic" si intende:

A) Il sistema risponde proporzionalmente alla quantità di richieste in ingresso B) Il sistema è disponibile anche in caso di guasto parziale C) Il sistema è in grado di consumare più o meno risorse per ottenere maggiori prestazioni o seguire un calo delle richieste D) Il sistema consuma dati da diverse fonti di ingresso

30. Le Conflict-Free Replicated Data-Types sono strutture dati particolarmente utili in alcuni tipi di applicazioni distribuite. La loro caratteristica è di fornire la garanzia che:

A) Modifiche fatte su nodi differenti non appaiono mai contemporanee B) Modifiche fatte su nodi differenti possono essere riordinate in sequenza causale C) Modifiche fatte contemporaneamente su nodi differenti sono sempre riconciliabili D) Modifiche fatte contemporaneamente su nodi differenti possono essere individuate e segnalate come conflitto

Parte 2: Domande Teoriche (3 punti ciascuna)

1. Descrivere le quattro condizioni di Coffman necessarie per un deadlock. Spiegare come la loro conoscenza può aiutare a prevenire situazioni di deadlock in un sistema concorrente.

2. Confrontare il paradigma di programmazione concorrente con quello distribuito, evidenziando similitudini e differenze. Fare almeno un esempio pratico per ciascun paradigma.

3. Spiegare le caratteristiche principali del modello ad attori. In quale contesto è particolarmente vantaggioso utilizzare questo paradigma rispetto ad altri approcci?

Parte 3: Esercizi Pratici (10 punti ciascuno)

Esercizio 1: Stream e Parallelismo

Dato il seguente codice che elabora una lista di numeri in modo sequenziale:

```
List<Integer> numbers = IntStream.rangeClosed(1, 10_000_000)
    .boxed()
    .collect(Collectors.toList());
```

```
long start = System.currentTimeMillis();
long count = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .filter(n -> n % 10 == 0)
    .count();
long end = System.currentTimeMillis();

System.out.println("Risultato: " + count);
System.out.println("Tempo impiegato: " + (end - start) + " ms");
```

1. Modificare il codice per utilizzare uno stream parallelo.
2. Aggiungere un metodo che calcoli il Blocking Factor dell'algoritmo.
3. Implementare una versione che permetta di specificare il grado di parallelismo.
4. Spiegare quale versione ci si aspetta sia più efficiente e perché.

Esercizio 2: Programmazione Concorrente

Implementare un sistema di produttore-consumatore con le seguenti caratteristiche:

1. Utilizzare una `BlockingQueue` per gestire la comunicazione tra produttori e consumatori.
2. Creare 3 produttori che generano numeri casuali e li inseriscono nella coda.
3. Creare 2 consumatori che prelevano i numeri dalla coda e calcolano il loro quadrato.
4. Ogni produttore deve generare 100 numeri casuali e poi terminare.
5. Il programma deve terminare quando tutti i produttori hanno terminato e la coda è vuota.
6. Stampare statistiche sull'elaborazione (tempo totale, numero di elementi elaborati da ciascun consumatore).

Esercizio 3: Programmazione di Rete

Implementare un semplice server di chat con le seguenti caratteristiche:

1. Il server deve accettare connessioni da più client contemporaneamente.
2. Quando un client invia un messaggio, il server deve inoltrarlo a tutti gli altri client connessi.
3. Quando un client si connette o si disconnette, tutti gli altri client devono essere avvisati.
4. Implementare una funzionalità di "nickname" per identificare i client.
5. Il server deve gestire correttamente la disconnessione improvvisa di un client.

Bonus: Implementare la funzionalità di "stanze" di chat, permettendo ai client di unirsi e lasciare diverse stanze.

Soluzioni

Parte 1: Domande a Risposta Multipla

1. C
2. B
3. A, C, E, F
4. C
5. A, D, E
6. A, C, D
7. D
8. 1-a, 2-b, 3-c, 4-d, 5-e
9. C
10. B, C
11. B
12. C
13. C
14. D
15. B
16. B, C, E, F
17. B
18. A, C, E
19. B
20. C
21. C
22. A, D
23. D
24. C
25. B
26. A
27. A, C
28. C, D
29. C
30. C

Parte 2: Domande Teoriche (Traccia delle risposte)

1. Condizioni di Coffman

Le quattro condizioni di Coffman necessarie per un deadlock sono:

1. **Mutua esclusione**: le risorse non possono essere condivise simultaneamente.
2. **Possesso e attesa** (Resource holding): un processo mantiene il possesso di almeno una risorsa mentre attende di acquisirne altre.
3. **No preemption**: le risorse non possono essere rilasciate forzatamente.

4. **Attesa circolare**: esiste una catena circolare di processi, ognuno dei quali è in attesa di una risorsa detenuta dal processo successivo nella catena.

Per prevenire deadlock, è sufficiente negare almeno una di queste condizioni:

- **Mutua esclusione**: usare algoritmi lock-free o wait-free per accedere alle risorse.
- **Possesso e attesa**: acquisire tutte le risorse in una transazione unica.
- **No preemption**: permettere il rilascio forzato delle risorse.
- **Attesa circolare**: imporre un ordinamento globale nell'acquisizione delle risorse.

2. Programmazione concorrente vs distribuita

Programmazione concorrente:

- Gestisce più linee di esecuzione sulla stessa macchina
- Condivide risorse come memoria e processore
- Obiettivo: migliorare l'utilizzo delle risorse di un singolo sistema
- Esempio: un'applicazione di elaborazione immagini che suddivide il lavoro tra più thread

Programmazione distribuita:

- Gestisce processi su macchine diverse
- Comunica tramite scambio di messaggi sulla rete
- Obiettivo: scalabilità, affidabilità, diffusione geografica
- Esempio: un sistema di chat dove i messaggi sono inoltrati tra diversi server regionali

Similitudini:

- Entrambi devono gestire problemi di sincronizzazione
- Entrambi devono preoccuparsi di fallimenti parziali
- Entrambi utilizzano asincronia per migliorare l'efficienza

3. Modello ad attori

Il modello ad attori è un paradigma di programmazione concorrente in cui le unità base di elaborazione sono gli attori. Caratteristiche principali:

- Ogni attore ha uno stato privato, inaccessibile dall'esterno
- Gli attori comunicano esclusivamente tramite messaggi asincroni
- Un attore può: modificare il proprio stato, creare nuovi attori, inviare messaggi, cambiare comportamento
- All'interno dell'attore non c'è concorrenza, ma gli attori sono concorrenti tra loro
- Il fallimento è considerato un comportamento normale e gestibile

Il modello ad attori è particolarmente vantaggioso in:

- Sistemi altamente concorrenti con molte componenti indipendenti
- Sistemi distribuiti che richiedono elevata tolleranza ai guasti
- Applicazioni che devono scalare linearmente aggiungendo nodi
- Sistemi in cui la semplicità del modello di comunicazione è più importante della performance pura

Esempi di utilizzo: sistemi di telecomunicazione, servizi cloud, sistemi IoT, applicazioni di streaming in tempo reale.

Parte 3: Esercizi Pratici (Traccia delle soluzioni)

Esercizio 1: Stream e Parallelismo

```
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class StreamParallelDemo {

    // Versione sequenziale
    public static long countSequential(List<Integer> numbers) {
        long start = System.currentTimeMillis();
        long count = numbers.stream()
            .filter(n -> n % 2 == 0)
            .map(n -> n * n)
            .filter(n -> n % 10 == 0)
            .count();
        long end = System.currentTimeMillis();

        System.out.println("Sequenziale - Risultato: " + count);
        System.out.println("Sequenziale - Tempo impiegato: " + (end - start)
+ " ms");
        return count;
    }

    // Versione parallela
    public static long countParallel(List<Integer> numbers) {
        long start = System.currentTimeMillis();
        long count = numbers.parallelStream()
            .filter(n -> n % 2 == 0)
            .map(n -> n * n)
            .filter(n -> n % 10 == 0)
            .count();
        long end = System.currentTimeMillis();

        System.out.println("Parallelo - Risultato: " + count);
        System.out.println("Parallelo - Tempo impiegato: " + (end - start) +
```

```
" ms");
        return count;
    }

    // Calcolo del Blocking Factor
    public static double calculateBlockingFactor(List<Integer> numbers) {
        // Un modo semplice è misurare il rapporto tra tempo di CPU e tempo
totale
        // BF = 0 significa che la CPU è sempre occupata
        // BF = 1 significa che la CPU è sempre in attesa di I/O
        // Simuliamo una misurazione
        return 0.1; // Predominanza di operazioni CPU-bound
    }

    // Versione con parallelismo personalizzato
    public static long countCustomParallel(List<Integer> numbers, int
parallelism) {
        ForkJoinPool customPool = new ForkJoinPool(parallelism);
        long start = System.currentTimeMillis();
        long count;

        try {
            count = customPool.submit(() ->
                numbers.parallelStream()
                    .filter(n -> n % 2 == 0)
                    .map(n -> n * n)
                    .filter(n -> n % 10 == 0)
                    .count()
            ).get();
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            customPool.shutdown();
        }

        long end = System.currentTimeMillis();

        System.out.println("Custom Parallelo (" + parallelism + " thread) -
Risultato: " + count);
        System.out.println("Custom Parallelo (" + parallelism + " thread) -
Tempo impiegato: " + (end - start) + " ms");
        return count;
    }

    public static void main(String[] args) {
        // Genera la lista di numeri
        List<Integer> numbers = IntStream.rangeClosed(1, 10_000_000)
            .boxed()
            .collect(Collectors.toList());
```

```
        // Calcola il Blocking Factor
        double bf = calculateBlockingFactor(numbers);
        System.out.println("Blocking Factor stimato: " + bf);

        // Calcola il parallelismo ottimale
        int cores = Runtime.getRuntime().availableProcessors();
        int optimalThreads = (int) Math.ceil(cores / (1 - bf));
        System.out.println("Numero di core: " + cores);
        System.out.println("Parallelismo ottimale stimato: " +
optimalThreads);

        // Esegui le varie versioni
        countSequential(numbers);
        countParallel(numbers);
        countCustomParallel(numbers, optimalThreads);

        // Con un BF basso (operazioni CPU-bound), ci aspettiamo che il
parallelismo
        // ottimale sia vicino al numero di core disponibili
    }
}
```

Esercizio 2: Programmazione Concorrente

```
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicInteger;

public class ProducerConsumerDemo {

    private static final int NUM_PRODUCERS = 3;
    private static final int NUM_CONSUMERS = 2;
    private static final int NUMBERS_PER_PRODUCER = 100;
    private static final int QUEUE_CAPACITY = 50;

    public static void main(String[] args) throws InterruptedException {
        // Crea la coda condivisa
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>
(QUEUE_CAPACITY);

        // CountDownLatch per aspettare che tutti i produttori finiscano
        CountDownLatch producerLatch = new CountDownLatch(NUM_PRODUCERS);

        // Flag per segnalare ai consumatori di continuare a lavorare
        AtomicInteger[] processedCount = new AtomicInteger[NUM_CONSUMERS];
        for (int i = 0; i < NUM_CONSUMERS; i++) {
```

```
            processedCount[i] = new AtomicInteger(0);
        }

        // Avvia il timer
        long startTime = System.currentTimeMillis();

        // Crea e avvia i produttori
        for (int i = 0; i < NUM_PRODUCERS; i++) {
            int producerId = i;
            new Thread(() -> {
                try {
                    Random random = new Random();
                    for (int j = 0; j < NUMBERS_PER_PRODUCER; j++) {
                        int number = random.nextInt(100);
                        queue.put(number);
                        System.out.println("Produttore " + producerId + " ha
generato: " + number);
                        // Simula un po' di lavoro
                        Thread.sleep(random.nextInt(10));
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    producerLatch.countDown();
                    System.out.println("Produttore " + producerId + " ha
terminato.");
                }
            }).start();
        }

        // Crea e avvia i consumatori
        Thread[] consumers = new Thread[NUM_CONSUMERS];
        for (int i = 0; i < NUM_CONSUMERS; i++) {
            int consumerId = i;
            consumers[i] = new Thread(() -> {
                try {
                    while (true) {
                        // Se tutti i produttori hanno terminato e la coda è
vuota, esci
                        if (producerLatch.getCount() == 0 &&
queue.isEmpty()) {
                            break;
                        }

                        Integer number = queue.poll();
                        if (number != null) {
                            int squared = number * number;
                            System.out.println("Consumatore " + consumerId +
" ha elaborato: " +
                                               number + " -> " + squared);
```

```
                            processedCount[consumerId].incrementAndGet();
                            // Simula un po' di lavoro
                            Thread.sleep(20);
                        }
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    System.out.println("Consumatore " + consumerId + " ha
terminato.");
                }
            });
            consumers[i].start();
        }

        // Attendi che tutti i consumatori terminino
        for (Thread consumer : consumers) {
            consumer.join();
        }

        // Calcola il tempo totale
        long endTime = System.currentTimeMillis();
        long totalTime = endTime - startTime;

        // Stampa le statistiche
        System.out.println("\n--- Statistiche ---");
        System.out.println("Tempo totale di esecuzione: " + totalTime + "
ms");
        int totalProcessed = 0;
        for (int i = 0; i < NUM_CONSUMERS; i++) {
            int count = processedCount[i].get();
            totalProcessed += count;
            System.out.println("Consumatore " + i + " ha elaborato " + count
+ " elementi");
        }
        System.out.println("Totale elementi elaborati: " + totalProcessed);
    }
}
```

Esercizio 3: Programmazione di Rete

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.*;

public class ChatServer {
```

```
    private static final int PORT = 8080;
    private final Set<ClientHandler> clients =
ConcurrentHashMap.newKeySet();
    private final Map<String, Set<ClientHandler>> rooms = new
ConcurrentHashMap<>();

    public static void main(String[] args) {
        new ChatServer().start();
    }

    public void start() {
        System.out.println("Server di chat avviato sulla porta " + PORT);

        // Crea una stanza generale
        rooms.put("generale", ConcurrentHashMap.newKeySet());

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Nuova connessione da " +
clientSocket.getInetAddress());

                ClientHandler clientHandler = new
ClientHandler(clientSocket, this);
                clients.add(clientHandler);
                new Thread(clientHandler).start();
            }
        } catch (IOException e) {
            System.err.println("Errore del server: " + e.getMessage());
        }
    }

    public void broadcast(String message, ClientHandler sender) {
        System.out.println("Broadcasting: " + message);
        for (ClientHandler client : clients) {
            if (client != sender) {
                client.sendMessage(message);
            }
        }
    }

    public void sendToRoom(String roomName, String message, ClientHandler
sender) {
        System.out.println("Messaggio nella stanza " + roomName + ": " +
message);
        Set<ClientHandler> roomClients = rooms.get(roomName);
        if (roomClients != null) {
            for (ClientHandler client : roomClients) {
                if (client != sender) {
                    client.sendMessage("[" + roomName + "] " + message);
```

```
                }
            }
        }
    }

    public void joinRoom(String roomName, ClientHandler client) {
        rooms.computeIfAbsent(roomName, k -> ConcurrentHashMap.newKeySet());
        rooms.get(roomName).add(client);
        broadcast(client.getNickname() + " si è unito alla stanza " +
roomName, client);
    }

    public void leaveRoom(String roomName, ClientHandler client) {
        Set<ClientHandler> roomClients = rooms.get(roomName);
        if (roomClients != null) {
            roomClients.remove(client);
            broadcast(client.getNickname() + " ha lasciato la stanza " +
roomName, client);
        }
    }

    public void removeClient(ClientHandler client) {
        clients.remove(client);
        for (Set<ClientHandler> roomClients : rooms.values()) {
            roomClients.remove(client);
        }
        broadcast(client.getNickname() + " si è disconnesso", null);
    }

    public Set<String> getRooms() {
        return rooms.keySet();
    }

    private static class ClientHandler implements Runnable {
        private final Socket socket;
        private final ChatServer server;
        private PrintWriter out;
        private BufferedReader in;
        private String nickname = "Anonimo";
        private String currentRoom = "generale";

        public ClientHandler(Socket socket, ChatServer server) {
            this.socket = socket;
            this.server = server;
        }

        public String getNickname() {
            return nickname;
        }
```

```
        @Override
        public void run() {
            try {
                out = new PrintWriter(socket.getOutputStream(), true);
                in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

                // Richiedi il nickname
                out.println("Benvenuto al server di chat! Inserisci il tuo
nickname:");
                nickname = in.readLine();

                // Unisciti alla stanza generale
                server.joinRoom("generale", this);

                // Invia un messaggio di benvenuto
                sendMessage("Benvenuto, " + nickname + "! Sei nella stanza
'generale'.");
                sendMessage("Comandi disponibili: /join [stanza], /leave
[stanza], /rooms, /nick [nuovonick], /quit");

                // Avvisa gli altri
                server.broadcast(nickname + " si è connesso", this);

                String inputLine;
                while ((inputLine = in.readLine()) != null) {
                    if (inputLine.startsWith("/")) {
                        // Comando
                        processCommand(inputLine);
                    } else {
                        // Messaggio normale
                        server.sendToRoom(currentRoom, nickname + ": " +
inputLine, this);
                    }
                }
            } catch (IOException e) {
                System.err.println("Errore con client " + nickname + ": " +
e.getMessage());
            } finally {
                try {
                    socket.close();
                } catch (IOException e) {
                    // Ignora
                }
                server.removeClient(this);
            }
        }

        private void processCommand(String command) {
            String[] parts = command.split(" ", 2);
```

```
            String cmd = parts[0].toLowerCase();

            switch (cmd) {
                case "/join":
                    if (parts.length > 1) {
                        String newRoom = parts[1];
                        server.leaveRoom(currentRoom, this);
                        server.joinRoom(newRoom, this);
                        currentRoom = newRoom;
                        sendMessage("Ti sei unito alla stanza " + newRoom);
                    } else {
                        sendMessage("Utilizzo: /join [stanza]");
                    }
                    break;

                case "/leave":
                    if (parts.length > 1) {
                        String roomToLeave = parts[1];
                        if (!roomToLeave.equals("generale")) {
                            server.leaveRoom(roomToLeave, this);
                            if (currentRoom.equals(roomToLeave)) {
                                currentRoom = "generale";
                                server.joinRoom("generale", this);
                                sendMessage("Sei tornato nella stanza
generale");
                            }
                        } else {
                            sendMessage("Non puoi lasciare la stanza
generale");
                        }
                    } else {
                        sendMessage("Utilizzo: /leave [stanza]");
                    }
                    break;

                case "/rooms":
                    sendMessage("Stanze disponibili: " + String.join(", ",
server.getRooms()));
                    break;

                case "/nick":
                    if (parts.length > 1) {
                        String oldNick = nickname;
                        nickname = parts[1];
                        server.broadcast(oldNick + " ha cambiato nickname in
" + nickname, null);
                        sendMessage("Hai cambiato nickname in " + nickname);
                    } else {
                        sendMessage("Utilizzo: /nick [nuovonick]");
                    }
```

```
                    break;

                case "/quit":
                    try {
                        sendMessage("Arrivederci!");
                        socket.close();
                    } catch (IOException e) {
                        // Ignora
                    }
                    break;

                default:
                    sendMessage("Comando sconosciuto: " + cmd);
                    break;
            }
        }

        public void sendMessage(String message) {
            out.println(message);
        }
    }
}
```