MEGA RACCOLTA Quiz Paradigmi di Programmazione

Prof. Michele Mauro - Università di Padova

Raccolta completa basata su appunti, materiali del corso e appelli precedenti

---

1. JAVA FUNDAMENTALS & LAMBDA EXPRESSIONS

1.1 Lambda Expressions e Functional Interfaces

Q1: Quale delle seguenti affermazioni riguardante le Lambda Expression è vera?

- a) Rendono Java un linguaggio funzionale perché possono essere passate come parametri
- b) **Non rendono Java un linguaggio funzionale perché non sono una entità del linguaggio, ma solo una convenienza sintattica risolta dal compilatore** ✅
- c) Non rendono Java un linguaggio funzionale perché non sono facilmente componibili
- d) Rendono Java un linguaggio funzionale perché permettono di astrarre sul comportamento

Q2: Una lambda expression in Java:

- a) Ha un tipo proprio definito dal compilatore
- b) **È solo una sintassi breve per implementare interfacce SAM (Single Abstract Method)** ✅
- c) Può essere utilizzata solo con Stream API
- d) Crea automaticamente un thread separato

Q3: Il metodo `forEach` applicato a una Collection:

- a) È sempre più efficace di un ciclo for tradizionale
- b) **Utilizza internamente un Iterator per attraversare gli elementi** ✅
- c) Modifica la collezione originale
- d) È disponibile solo per List e Set

1.2 Switch Case e Pattern Matching

Q4: Quale di queste caratteristiche è propria della sintassi switch-case come espressione:

- a) I risultati devono essere tutti valori della stessa interfaccia
- b) È possibile il fall-through da un caso all'altro
- c) **L'elenco delle opzioni deve essere esaustivo** ✅
- d) Ogni caso deve produrre un risultato diverso

1.3 Generics e Type System

Q5: I tipi generici in Java:

- a) Sono mantenuti a runtime (reificazione)
- b) **Vengono cancellati dal compilatore (type erasure)** ✅
- c) Possono utilizzare tipi primitivi come parametri
- d) Non supportano wildcards

Q6: In una definizione di metodo generico, cosa rappresenta `<T extends Comparable<T>>`?

- a) T deve essere una sottoclasse di Comparable
- b) **T deve implementare l'interfaccia Comparable con se stesso come parametro** ✅
- c) T può essere qualsiasi tipo
- d) T deve avere un costruttore di default

---

2. STREAM API

2.1 Funzionalità Base degli Stream

Q7: 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, operando una scelta ben definita. **In quali casi può essere necessario modificare il comportamento di default?**

- 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** ✅

Q8: Nella implementazione degli Stream della libreria standard, che vantaggio si ottiene dal fatto che la API consente di costruire la catena di elaborazione separatamente dalla sua esecuzione?

- a) **L'implementazione può analizzare le operazioni della catena, e prendere decisioni su come applicarle in funzione delle loro caratteristiche** ✅
- b) L'implementazione può sempre sapere se dovrà eseguire un numero finito o meno di elaborazioni
- c) L'implementazione può analizzare le operazioni della catena, e decidere se eseguirle parallelamente o in serie

Q9: Le operazioni intermedie sugli Stream:

- a) Vengono eseguite immediatamente quando chiamate
- b) **Non devono interferire (modificare elementi dello stream)** ✅
- c) **Nella maggior parte dei casi non devono avere stato interno** ✅
- d) **Devono essere stateless per garantire il parallelismo** ✅

2.2 Collector Interface

Q10: 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) Mantiene il parallelismo dello Stream
- c) **Gestisce un accumulatore mutabile, che riduce la pressione sulla Garbage Collection** ✅
- d) Combina i risultati intermedi più velocemente

---

3. PROGRAMMAZIONE CONCORRENTE

3.1 Thread e Sincronizzazione

Q11: Associare ciascuna struttura di gestione della concorrenza al suo ambito di applicazione:

- **Semaphore → A) Gestione di un insieme omogeneo di risorse** ✅
- **Lock → B) Gestione esplicita, senza legami sintattici, del blocco e dello sblocco della sezione critica** ✅
- **synchronized → E) Gestione della concorrenza tramite la struttura sintattica del codice** ✅
- **condition → D) Gestione dell'accesso alla stessa sezione critica in condizioni di blocco e/o sblocco differenti** ✅
- **object::wait() → C) Gestione esplicita dell'accesso ad un singolo oggetto** ✅

Q12: Una variabile di tipo ThreadLocal<T>:

- a) **Possiede un valore differente per ogni Thread che vi accede** ✅
- b) Permette a più Thread di accedere rapidamente al valore che contiene
- c) Un solo Thread per volta può accedere al valore contenuto
- d) Più Thread possono accedere allo stesso valore senza interferire tra loro

Q13: Gli stati di un thread includono:

- a) **NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED** ✅
- b) CREATED, RUNNING, SUSPENDED, DEAD
- c) INIT, ACTIVE, SLEEPING, FINISHED
- d) START, EXECUTE, PAUSE, STOP

---

4. PROGRAMMAZIONE DISTRIBUITA

4.1 Networking e Socket

Q14: La comunicazione su Datagram presenta alcuni vantaggi rispetto ai Socket, ma è afflitta dai seguenti svantaggi:

- a) Le due parti devono concordare l'encoding delle stringhe
- b) Deve essere definito un protocollo per riconoscere inizio e fine dei messaggi
- c) **La probabilità di successo della comunicazione diminuisce con il crescere della lunghezza del messaggio** ✅
- d) Le due parti devono inviare i dati il più velocemente possibile

Q15: I Channel in Java NIO:

- a) Sono sempre sincroni
- b) **Permettono operazioni I/O asincrone** ✅
- c) **Richiedono CompletionHandler per gestire eventi asincroni** ✅
- d) **Supportano l'attachment per passare contesto tra operazioni** ✅

4.2 RPC e Sistemi Distribuiti

Q16: Remote Procedure Call (RPC):

- a) **Rende trasparente la localizzazione del codice chiamato** ✅
- b) **Utilizza marshalling per convertire parametri in messaggi** ✅
- c) **In linguaggi OOP diventa RMI (Remote Method Invocation)** ✅
- d) Garantisce sempre la delivery dei messaggi

---

5. FALLACIES OF DISTRIBUTED COMPUTING

Q17: La fallacia "Latency is zero" è ancora rilevante perché:

- a) L'aumento dei nodi della rete ha compensato il miglioramento tecnologico
- b) Le tecnologie hanno eliminato il problema ma le esigenze di concorrenza l'hanno reintrodotto
- c) **Dipende da una grandezza fisica (velocità della luce)** ✅
- d) I protocolli moderni non ottimizzano per la latenza

Q18: La fallacia "The network is reliable" implica che:

- a) I guasti hardware sono l'unica fonte di inaffidabilità
- b) **Il software distribuito deve gestire connessioni interrotte e messaggi persi** ✅
- c) **È necessario bilanciare il costo della ridondanza con il rischio di failure** ✅
- d) La rete è intrinsecamente sicura

Q19: La fallacia "Bandwidth is infinite" comporta che:

- a) La banda disponibile non è mai un problema
- b) **I dati da trasferire aumentano velocemente quanto la banda disponibile** ✅
- c) **I protocolli privilegiano affidabilità rispetto a economia di trasmissione** ✅
- d) La banda è sempre costante

Q20: Le 8 Fallacies of Distributed Computing sono:

1. **The network is reliable**
2. **Latency is zero**
3. **Bandwidth is infinite**
4. **The network is secure**
5. **Topology doesn't change**
6. **There is one administrator**
7. **Transport cost is zero**
8. **The network is homogeneous**

(Nota: esiste una nona fallacia "Time is ubiquitous" aggiunta da Deutsch)

---

6. CAP THEOREM & CONSENSO

6.1 CAP Theorem

Q21: Il teorema CAP afferma che un sistema distribuito, in caso di partizione dei suoi nodi, deve scegliere tra:

- a) Consistenza e latenza
- b) Correttezza ed efficienza
- c) **Consistenza e disponibilità** ✅
- d) Certezza e latenza delle risposte

Q22: CAP Theorem definisce:

- **C (Consistency): Ogni lettura riceve il valore più recente o un errore** ✅
- **A (Availability): Ogni richiesta riceve una risposta valida (non necessariamente l'ultimo valore)** ✅
- **P (Partition Tolerance): Il sistema funziona anche se la rete viene partizionata** ✅

Q23: Se in un sistema distribuito i nodi non trovano un consenso sullo stato del sistema, può accadere che:

- a) **Le risposte del sistema siano incoerenti e dipendano da quale nodo viene contattato** ✅
- b) Le risposte siano molteplici e conflittuali perché raccolgono dati da più nodi
- c) Le risposte siano inefficienti per race condition
- d) Le risposte non siano disponibili perché gli stati si annullano

6.2 PACELC e Estensioni

Q24: L'estensione PACELC del CAP Theorem considera:

- **P:** in caso di Partizione, scegli tra **A (Availability) e C (Consistency)**
- **E:** Else (altrimenti), scegli tra **L (Latency) e C (Consistency)**

Q25: I sistemi NoSQL generalmente si orientano verso:

- a) PC/EC (priorità alla consistenza)
- b) **PA/EL (priorità a disponibilità e bassa latenza)** ✅
- c) Solo C (solo consistenza)
- d) Tutti e tre insieme

6.3 Algoritmi di Consenso

Q26: L'algoritmo PAXOS:

- a) **È basato su un protocollo a tre fasi** ✅
- b) **Gestisce ruoli: leader, votanti, ascoltatori, proponente, client** ✅
- c) **Il proponente inoltra richieste ai votanti per raggiungere quorum** ✅
- d) Non può gestire guasti bizantini

Q27: RAFT rispetto a PAXOS:

- a) **È più comprensibile e suddivide il problema in sottoproblemi** ✅
- b) **Si basa sulla replicazione di una macchina a stati** ✅
- c) **Enfatizza il ruolo del leader e la sua elezione** ✅
- d) È meno efficiente

---

7. CRDT & CALM THEOREM

7.1 Conflict-Free Replicated Data Types

Q28: Le CRDT (Conflict-Free Replicated Data Types):

- a) **Garantiscono la riconciliazione di scritture concorrenti** ✅
- b) **Permettono convergenza su un valore che include tutte le modifiche** ✅
- c) Richiedono un coordinator centrale
- d) **Esempi: Grow-only Counter, 2-Phase Set, Last-Write-Wins Set** ✅

Q29: Operational Transformation vs CRDT:

- a) **OT: ogni nodo trasforma le modifiche ricevute per applicarle al suo stato** ✅
- b) **OT: usato in Google Docs ma manca di generalità** ✅
- c) **CRDT: più semplice da implementare e più generale** ✅
- d) Sono equivalenti in termini di prestazioni

7.2 CALM Theorem

Q30: CALM (Consistency As Logical Monotonicity) afferma che:

- a) **I programmi CP (del CAP) sono esprimibili in logica monotona** ✅
- b) **Un programma è logicamente monotono se all'aumentare della dimensione il risultato non cambia** ✅
- c) Tutti i programmi distribuiti sono monotoni
- d) **È un risultato costruttivo ma la classe di programmi è poco popolata** ✅

Q31: Esempi di sistemi logicamente monotoni/non monotoni:

- **Monotono: Sistema che individua deadlock (una volta trovato, non cambia)** ✅
- **Non monotono: Garbage Collection distribuita (aggiungendo nodi può cambiare risultato)** ✅

---

8. REACTIVE PROGRAMMING

8.1 Reactive Extensions

Q32: Lo scopo delle Reactive Extensions è:

- a) Fornire una API per definire elaborazioni di sequenze di oggetti
- b) **Fornire una semantica per definire elaborazioni asincrone di sequenze di oggetti** ✅
- c) Fornire componenti per l'elaborazione distribuita di stream
- d) Fornire un modello di esecuzione parallele di insiemi

Q33: Nelle Reactive Extensions, quali operazioni NON è necessario (o possibile) specificare per elaborare gli oggetti emessi da un Observable:

- a) Il comportamento alla ricezione di un oggetto
- b) **Il numero di oggetti che l'Observable è autorizzato ad inviare** ✅
- c) **Il comportamento alla richiesta di separazione di uno stream parallelo** ✅
- d) Il comportamento al termine dello stream
- e) Il comportamento alla ricezione di un errore

8.2 Reactive Manifesto

Q34: Il Reactive Manifesto definisce le caratteristiche dei sistemi reattivi:

- a) **Pronti alla risposta (Responsive)** ✅
- b) **Resilienti (Resilient)** ✅
- c) **Elastici (Elastic)** ✅
- d) **Orientati ai messaggi (Message Driven)** ✅

Q35: Subject in RxJava:

- a) **Può ascoltare più Observable ed essere osservato** ✅
- b) **Può manipolare i dati di un Observable prima di "riemetterli"** ✅
- c) È solo un tipo di Observable
- d) Non può gestire errori

8.3 Virtual Threads vs Reactive

Q36: Secondo Brian Goetz, Virtual Threads vs Reactive Programming:

- a) **"Loom is going to kill Reactive Programming"** ✅
- b) **Reactive Programming era una tecnologia di transizione** ✅
- c) **Virtual Threads permettono prestazioni elevate con codice tradizionale** ✅
- d) Reactive Programming rimarrà dominante

---

9. FRAMEWORKS & PARADIGMI

9.1 Vantaggi e Svantaggi dei Framework

Q37: Vantaggi dei framework per applicazioni distribuite:

- a) **Maggiore sicurezza perché i dettagli sono gestiti da persone più competenti** ✅
- b) **Aggiornamento continuo che apporta benefici automatici** ✅
- c) Estrema efficienza nello sfruttare peculiarità hardware
- d) **Facilità di realizzazione perché i dettagli dei protocolli sono nascosti** ✅
- e) **Facilità perché le parti strutturali sono già implementate** ✅

Q38: Svantaggi dei framework:

- a) **Gli sviluppatori potrebbero non avere le nostre stesse priorità** ✅
- b) **Il nostro caso d'uso potrebbe non essere ben supportato** ✅
- c) **Possiamo incontrare errori imprevedibili** ✅
- d) **L'evoluzione può richiedere costi di sviluppo non controllabili** ✅

9.2 Paradigmi di Programmazione

Q39: Una classe Java dichiarata abstract:

- a) È visibile solo dalle classi che la estendono
- b) È visibile da tutte le classi dello stesso package
- c) **abstract non è un modificatore di visibilità** ✅
- d) È visibile da qualsiasi classe

Q40: L'ereditarietà in Java:

- a) Una interfaccia può implementare una sola altra interfaccia
- b) **A causa dei default methods, è possibile causare un Diamond Problem** ✅
- c) **Una interfaccia può ereditare da un'altra interfaccia** ✅
- d) **Una classe può implementare più interfacce** ✅
- e) **Una classe può ereditare da una sola altra classe** ✅

---

10. DOMANDE PRATICHE E APPLICATIVE

10.1 Ordine di Esecuzione in Java

Q41: Date le seguenti classi, ordinare secondo la sequenza di esecuzione:

java

```java
class Foo {
    Foo(int a) { //Costruttore Foo }
}
class Bar extends Foo {
    static { //Inizializzatore statico }
    { //Inizializzatore }
    Bar(int a, String b) {
        super(a); //Costruttore Bar
    }
}
```

Ordine corretto:

1. **Inizializzatore statico** ✅
2. **Costruttore Foo** ✅
3. **Inizializzatore** ✅
4. **Costruttore Bar** ✅

10.2 Monotonicità e Logica

Q42: Un programma è "logicamente monotono" se:

- a) È sempre più veloce con più risorse
- b) **All'aumentare della dimensione del problema il risultato non cambia** ✅
- c) Utilizza sempre la stessa quantità di memoria
- d) Non ha effetti collaterali

---

11. DOMANDE AVANZATE E CASI STUDIO

11.1 Performance e Ottimizzazione

Q43: Il blocking factor di un algoritmo rappresenta:

- a) Il numero di thread bloccati
- b) **L'intensità del calcolo: BF=0 (occupa sempre CPU), BF=1 (sempre in I/O)** ✅
- c) Il tempo di attesa massimo
- d) La probabilità di deadlock

Q44: La formula per il numero ottimale di thread è:

- a) Numero di core + 1
- b) **#cores / (1 - BF)** ✅
- c) 2 × numero di core
- d) Dipende solo dalla RAM disponibile

11.2 Implementazioni Specifiche

Q45: Stream Gatherers (Java 22+):

- a) **Implementano l'interfaccia Gatherer con 4 funzioni** ✅
- b) **initializer, integrator, combiner, finisher** ✅
- c) **Facilitano operazioni intermedie stateful** ✅
- d) Sostituiscono completamente i Collector

11.3 Sistemi Distribuiti Avanzati

Q46: Back-pressure nei Reactive Streams:

- a) È un problema da evitare
- b) **Permette di comunicare quanti dati un componente può elaborare** ✅
- c) **Gestisce la velocità dei dati in ingresso** ✅
- d) È presente solo in RxJava

DOMANDE BONUS: DETTAGLI TECNICI

Q47: Progetti che implementano CRDT:

- a) **Akka (modulo Akka Data)** ✅
- b) **Riak** ✅
- c) **Redis Enterprise** ✅
- d) **Facebook Apollo** ✅

Q48: VertX framework caratteristiche:

- a) **Event-driven e asincrono** ✅
- b) **Modulare e compatto** ✅
- c) **Unopinionated** ✅
- d) **Supporta più linguaggi** ✅

Q49: RxJava Schedulers:

- `.io()` → **Per operazioni di I/O** ✅
- `.computation()` → **Per calcoli intensivi** ✅
- `.single()` → **Usa un singolo thread** ✅
- `.from(executor)` → **Usa l'Executor fornito** ✅

Q50: La nona fallacia (aggiunta da Deutsch):

- a) The cloud is infinite
- b) **Time is ubiquitous** ✅
- c) Memory is unlimited
- d) All nodes are identical

CONSIGLI PER L'ESAME

Argomenti ad Alta Frequenza:

1. **CAP Theorem e PACELC** - Quasi sempre presente
2. **Lambda Expressions vs Linguaggi Funzionali** - Domanda ricorrente
3. **Stream API e Parallelismo** - Molto probabile
4. **Fallacies of Distributed Computing** - Spesso 2-3 domande
5. **Reactive Extensions vs Virtual Threads** - Tendenza recente
6. **CRDT e Consenso** - Argomenti emergenti
7. **Framework pro/contro** - Domanda "filosofica" comune

Pattern delle Domande:

- **Definizioni precise** (CAP, CALM, Reactive Manifesto)
- **Confronti tecnologici** (RxJava vs Virtual Threads)
- **Ordinamento/associazione** (Thread states, execution order)
- **Identificazione casi d'uso** (quando usare parallelismo)
- **Vero/Falso multipli** (caratteristiche tecnologie)

Suggerimenti:

- Focalizzarsi sui **principi teorici** dietro le tecnologie
- Comprendere **trade-off e compromessi** (CAP, performance vs semplicità)
- Memorizzare **definizioni precise** dei teoremi
- Praticare **associazioni** (strutture dati → casi d'uso)
- Rivedere **appelli precedenti** per pattern delle domande