Domande per orale Paradigmi di Programmazione

Questo documento raccoglie una collezione di domande che sono state sentite durante l'esame orale di paradigmi di programmazione, ed è stato preparato con le slides fornite dalla Prof. Bodei nell' A.A. 2025/26. Il documento non raccoglie tutte le risposte a queste domande, e le domande che hanno le risposte potrebbero non essere del tutto corrette, a causa di mie sviste. Parte di queste risposte sono state scritte con l'aiuto di intelligenza artificiale. Verificare sempre il materiale sulle slides fornito dal docente!

Nel sommario trovate tutte le domande suddivise per macro-argomenti, e sotto, alle pagine indicate nel sommario, trovate delle risposte possibili per la maggior parte di queste domande.

Ricordo, per completezza, che l'orale di Paradigmi di Programmazione è strutturato con una prima parte scritta che dura 30 minuti (in cui vengono fatte 3 o 4 domande di teoria, di cui una contiene un'analisi di un codice in cui si chiede di descrivere quello che succede, oppure lo stato della memoria, oppure disegnare i Dispatch Vectors e l'iTable, oppure capire cosa succede in un pezzo di codice di programmazione concorrente.

Sommario

Lambda Calcolo

Cosa dice il teorema di Confluenza del Lambda Calcolo (teorema di Church Rosser)?

Il Teorema di Church-Rosser, noto anche come proprietà di confluenza, garantisce che il risultato di un calcolo non dipenda dall'ordine in cui vengono eseguite le riduzioni. Formalmente afferma che:

Se un termine M può essere ridotto in due modi diversi ottenendo i termini N e P, allora esiste sempre un termine comune Z a cui sia N che P possono essere ulteriormente ridotti. Geometricamente viene spesso rappresentato come la **'Proprietà del Rombo'** (Diamond Property). La conseguenza fondamentale di questo teorema è l'**unicità della Forma Normale**: se un termine ha una forma normale (ovvero un risultato finale), questa è unica, indipendentemente dalla strategia di riduzione scelta.

enunciare e spiegare le proprietà di progresso e conservazione per espressioni aritmetiche oppure per il lambda calcolo tipato

La **Type Safety** è data dalla somma di due teoremi:

1. **Progresso:** Un termine ben tipato non si blocca: o è un risultato finale (valore) o può avanzare.
2. **Conservazione:** Se un termine ben tipato fa un passo di calcolo, il termine risultante mantiene lo stesso tipo.

1. **Proprietà di Progresso (Progress):** Questa proprietà garantisce che un programma valido non si "incastri" in uno stato che non è finale. **Spiegazione:** Il Progresso ci assicura che se il *type checker* ha accettato il programma, la macchina

virtuale non si troverà mai di fronte a un'istruzione che non sa come eseguire, a meno che il programma non sia già terminato.

- **Contro-esempio (Violazione del Progresso):** Immagina di avere l'espressione true + 5. Se il sistema di tipi la accettasse, il programma si bloccherebbe a runtime perché non esiste una regola per sommare un booleano a un numero. Il teorema di Progresso garantisce che una situazione del genere ("Stuck state" che non è un valore) non accada mai per i termini ben tipati.

2. **Proprietà di Conservazione (Preservation):** Questa proprietà (spesso chiamata Subject Reduction) garantisce che il tipo di un termine non cambi mentre viene eseguito. **Spiegazione:** La Conservazione ci assicura che l'esecuzione del programma preserva la correttezza dei tipi. Se inizio con un'espressione che il compilatore mi dice essere un *Integer*, dopo aver fatto un calcolo intermedio, ciò che ottengo è ancora un *Integer*. Senza questa proprietà, il controllo dei tipi statico sarebbe inutile, perché un termine potrebbe "degenerare" in qualcosa di sbagliato durante l'esecuzione.

OCaml e interpreti

Spiegare la differenza tra type-checking statico e dinamico, e i rispettivi vantaggi e svantaggi.

1. Type-Checking Statico (Static Typing)

Il controllo dei tipi avviene **a tempo di compilazione** (Compile-time), ovvero *prima* che il programma venga eseguito.

Tabella di Confronto Rapido

Caratteristica	Statico (es. Java)	Dinamico (es. Python)
Quando controlla?	Durante la Compilazione	Durante l'Esecuzione
Se c'è errore?	Il programma non parte	Il programma crasha mentre gira
Prestazioni	Alte (codice ottimizzato)	Basse (overhead dei controlli)
Flessibilità	Bassa (rigido)	Alta (Duck Typing)
Rilevamento Bug	Precoce (Early detection)	Tardivo (Late detection)

Figura 1-Confronto typeChecking

Il compilatore analizza il codice sorgente e garantisce che non ci siano violazioni di tipo. Se c'è un errore (es. sommare un numero a una stringa), il programma **non viene nemmeno compilato** e non viene generato l'eseguibile.

- **Esempi:** Java, C, C++, Rust, Haskell.

Vantaggi

1. **Sicurezza (Safety):** Intercetta molte classi di bug banali molto prima che il software arrivi all'utente. "Well-typed programs cannot go wrong" (nel senso dei tipi).
2. **Prestazioni:** Poiché il compilatore conosce già i tipi, può generare codice macchina ottimizzato. Non c'è bisogno di controllare i tipi mentre il programma gira.

3. **Documentazione e Refactoring:** Le annotazioni di tipo aiutano i programmatori a capire cosa fa una funzione e permettono agli IDE di fare refactoring intelligenti e autocompletamento preciso.

Svantaggi

1. **Verbosità:** Spesso richiede di scrivere più codice (dichiarare esplicitamente i tipi), anche se l'inferenza di tipo moderna (come var in Java 10+) mitiga questo problema.
2. **Rigidità:** È più difficile scrivere codice generico o strutture dati molto flessibili senza usare costrutti complessi (come i Generics).
3. **Tempi di compilazione:** Il processo di verifica richiede tempo.

2. Type-Checking Dinamico (Dynamic Typing)

Il controllo dei tipi avviene **a tempo di esecuzione** (Runtime), ovvero *mentre* il programma sta girando.

Le variabili non hanno un tipo fisso; sono i **valori** (gli oggetti in memoria) ad avere un tipo. Il controllo viene fatto nel momento esatto in cui un'operazione viene eseguita. Se c'è un errore, il programma lancia un'eccezione e potenzialmente crasha in quel punto.

- **Esempi:** Python, JavaScript, Ruby, PHP.

Vantaggi

1. **Flessibilità:** È possibile scrivere funzioni che accettano qualsiasi cosa (Duck Typing: "Se cammina come un'anatra e starnazza come un'anatra, allora è un'anatra").
2. **Velocità di sviluppo:** Ideale per la prototipazione rapida. Scrivi meno codice ("boilerplate") e vedi subito il risultato.
3. **Semplicità:** Curva di apprendimento iniziale spesso più bassa.

Svantaggi

1. **Errori a Runtime:** Un bug di tipo potrebbe nascondersi per mesi e manifestarsi solo quando un utente esegue una specifica riga di codice raramente usata.
2. **Performance:** L'interprete deve controllare i tipi ogni volta che esegue un'operazione, rallentando l'esecuzione.
3. **Manutenibilità:** In progetti molto grandi (milioni di righe), non sapere di che tipo è una variabile rende difficile capire e modificare il codice senza rompere nulla.

nell'interprete qual' erano i ruoli del evt e exp

Cos’è la tail recursion e qual è il suo vantaggio? Quante chiamate ricorsive fa?

Una funzione si dice **Tail Recursive** (ricorsiva in coda) quando la chiamata ricorsiva è **l'ultimissima azione** eseguita dalla funzione stessa. Non deve esserci alcun calcolo o operazione in attesa che la chiamata ricorsiva ritorni un valore.

A. Ricorsione Standard (Non-Tail) Qui la moltiplicazione avviene *dopo* che la chiamata ricorsiva è ritornata.

Python

```python
def fattoriale_standard(n):
    if n == 0: return 1
    # La moltiplicazione (n *) avviene DOPO il ritorno di fattoriale_standard(n-1)
    return n * fattoriale_standard(n - 1)
```

B. Tail Recursion Qui usiamo un "accumulatore" per passare il risultato parziale. La chiamata è l'ultima cosa che succede.

Python

```python
def fattoriale_tail(n, accumulatore=1):
    if n == 0: return accumulatore
    # Nessuna operazione deve essere fatta dopo questa chiamata
    return fattoriale_tail(n - 1, n * accumulatore)
```

La Tail Recursion permette, quindi, di trasformare una funzione ricorsiva in un semplice ciclo (loop) a livello di codice macchina, mantenendo lo spazio di memoria costante $O(1)$ invece che lineare $O(N)$.

Come è fatto un compilatore (front end e back end)? Quali sono le fasi della compilazione nel Front-End?

La struttura di un compilatore è progettata modularmente per trasformare un programma scritto in un linguaggio di alto livello (Codice Sorgente) in un linguaggio di basso livello comprensibile dalla macchina (Codice Oggetto/Macchina), passando attraverso una rappresentazione intermedia.

Possiamo dividere questa architettura in due grandi macro-blocchi: il **Front-End** (Analisi) e il **Back-End** (Sintesi), collegati da una **Rappresentazione Intermedia (IR)**.

A. Front-End (Analisi)

- È la parte del compilatore che "capisce" il codice sorgente. È specifica per il linguaggio di programmazione (Java, C, Python, ecc.).
  Obiettivo: Verificare che il codice sia corretto (grammaticalmente e logicamente) e tradurlo in una rappresentazione intermedia.
  Se c'è un errore: Qui è dove vengono generati i messaggi di errore per il programmatore (es. "manca un punto e virgola", "variabile non dichiarata").

B. Intermediate Representation (IR)

- È il ponte tra i due mondi. È un codice neutro, né di alto livello (come Java) né di basso livello (come Assembly).
- **Vantaggio:** Permette di staccare il Front-End dal Back-End. Se vuoi creare un nuovo linguaggio, scrivi solo un nuovo Front-End e riusi il Back-End esistente (è il segreto del successo di **LLVM**).

C. Back-End (Sintesi)

È la parte che "genera" il codice finale. È specifica per la macchina di destinazione (Intel x86, ARM, MIPS).

- **Obiettivo:** Ottimizzare il codice intermedio e tradurlo in istruzioni binarie che la CPU può eseguire.

Il Front-End esegue un processo a cascata composto da quattro fasi principali. Vediamole usando come esempio l'istruzione: int position = initial + rate * 60;

Fase 1: Analisi Lessicale (Scanner)

Riassunto schematico

Fase	Input Principale	Cosa fa?	Output Principale
Lessicale	Codice Sorgente (Caratteri)	Riconosce le parole	Token
Sintattica	Token	Riconosce la struttura (Grammatica)	Syntax Tree (AST)
Semantica	AST	Riconosce il senso (Tipi e Scope)	AST Decorato / Symbol Table
Generazione IR	AST Decorato	Semplifica per il Back-End	Codice Intermedio

Il compilatore legge il flusso di caratteri del file sorgente e li raggruppa in unità significative chiamate **Token**.

- **Input:** i, n, t, , p, o, s... (caratteri grezzi)
- **Azione:** Elimina spazi bianchi e commenti. Riconosce le parole chiave e gli identificatori.
- **Output (Stream di Token):** [KEYWORD: int], [ID: position], [OP: =], [ID: initial], [OP: +], [ID: rate], [OP: *], [LITERAL: 60], [SEP: ;]

Fase 2: Analisi Sintattica (Parser)

Verifica che la sequenza di token rispetti la **grammatica** del linguaggio. Costruisce una struttura ad albero chiamata **Parse Tree** o **Abstract Syntax Tree (AST)**.

- **Input:** Stream di Token.
- **Azione:** Controlla l'ordine (es. "dopo un tipo ci deve essere un nome variabile"). Gestisce la precedenza degli operatori (la moltiplicazione vince sulla somma).
- **Output (AST):** Un albero gerarchico.

Fase 3: Analisi Semantica

Controlla che l'albero sintattico abbia "senso". Mentre la sintassi controlla la *forma*, la semantica controlla il *contenuto*.

- **Input:** AST.
- **Azione:**

- **Type Checking:** Sto cercando di sommare una stringa a un numero? (Errore).
- **Scope:** La variabile rate è stata dichiarata prima di essere usata?
- **Binding:** Collega ogni uso di variabile alla sua definizione.

- **Output:** AST Decorato (arricchito con informazioni sui tipi) e Symbol Table (tabella dei simboli popolata).

Fase 4: Generazione del Codice Intermedio (ICG)

Traduce l'AST decorato in un linguaggio lineare e astratto, simile all'Assembly ma con registri infiniti. Spesso si usa il **Three-Address Code (TAC)**.

- **Input:** AST Decorato.
- **Output (Esempio TAC):**

Java

In Java, Integer è sottotipo di Number. Che relazione c'è quindi tra List<integer> e List<Number>? Per quale motivo?

In Java, **non c'è alcuna relazione di sottotipo** tra List<Integer> e List<Number>.

Anche se Integer estende Number, le due liste sono considerate tipi **distinti e incompatibili**. In termini tecnici, si dice che i Generics in Java sono **INVARIANTI**.

```
List<Integer> interi = new ArrayList<>();
interi.add(10);

// IPOTESI: Se questo fosse permesso (Covarianza)...
List<Number> numeri = interi;

// ...allora potrei fare questo:
numeri.add(3.14); // LECITO per List<Number>, perché 3.14 è un Double (che è un Number)

// IL PROBLEMA:
// Ora la lista "interi" contiene un numero con la virgola (3.14)!
Integer n = interi.get(1); // CRASH a runtime (ClassCastException)
```

Figura 2-Type safety

Il motivo è garantire la **sicurezza dei tipi (Type Safety)** ed evitare errori a runtime. Se Java permettesse la covarianza (cioè se List<Integer> fosse sottotipo di List<Number>), si creerebbe una falla nel sistema che porterebbe alla corruzione della memoria.

Quali rischi si incorrono con un downcast esplicito e come mitigarli?

Il rischio principale è la rottura della type safety. Il controllo del downcast in Java viene fatto a runtime, e se l'oggetto che si trova nell Heap non è davvero un'istanza della classe verso cui si sta facendo il cast (o di una sua sottoclasse), la JVM lancia errore.

Esempio errore

```
Java

Object obj = "Sono una stringa"; // Tipo apparente: Object, Tipo reale: String
Integer numero = (Integer) obj;  // DOWNCAST ERRATO!
```

```java
Object obj = "Sono una stringa";

if (obj instanceof Integer) {
    Integer n = (Integer) obj; // Ora è sicuro al 100%
    System.out.println(n * 2);
} else {
    System.out.println("L'oggetto non è un numero!");
}
```

Correzione errore

scrivi codice di oggetti che siano in relazione di sottotipo strutturale , nominale, e oggetti che non lo siano

Il Sottotipo Nominale (Nominal Subtyping)

"Nel sistema di sottotipazione nominale, la relazione di sottotipo è **esplicita** e basata sui **nomi** delle classi o delle interfacce. Affinché il tipo A sia sottotipo del tipo B, deve essere stato dichiarato esplicitamente nel codice (usando parole chiave come extends o implements). Non importa se A e B hanno identici metodi e campi: se non c'è la dichiarazione formale di ereditarietà, per il compilatore sono tipi estranei. È il modello utilizzato da **Java**, C++, C#."

Il Sottotipo Strutturale (Structural Subtyping)

"Nel sistema di sottotipazione strutturale, la relazione è **implicita** e basata sulla **forma** (struttura) dell'oggetto. Affinché A sia sottotipo di B, è sufficiente che A contenga tutti i metodi e i campi pubblici richiesti da B (con firme compatibili). Non serve alcuna dichiarazione esplicita. Questo approccio è spesso riassunto dalla filosofia del *Duck Typing*: 'Se cammina come un'anatra e starnazza come un'anatra, allora è un'anatra'. È il modello utilizzato da **TypeScript**, Go, OCaml."

Vogliamo un oggetto che sia in grado di "Stampare".

Java

```java
// DEFINIZIONE DEL CONTRATTO (Il "Tipo padre")
interface Stampabile {
    void stampa();
}
```

B. Sottotipo Strutturale (Il caso "interessante")

Qui abbiamo una classe che ha la stessa struttura, ma non dichiara nulla.

Java

```java
class Foto {
    // Ha lo stesso identico metodo di Stampabile!
    public void stampa() {
        System.out.println("Sto stampando una foto...");
    }
}
```

A. Sottotipo Nominale (Java)

Qui la relazione esiste perché l'ho **scritta** (`implements`).

Java

```java
// RELAZIONE NOMINALE: ESISTE
class Documento implements Stampabile {
    public void stampa() {
        System.out.println("Sto stampando un documento...");
    }
}

// Utilizzo
Stampabile obj = new Documento(); // OK: Compila perfettamente.
```

In Java, la sottoclasse Foto non sarebbe sottotipo di Stampabile (perché non la implementa, quindi non c'è Nominal Subtyping), mentre in JavaScript si perché hanno la stessa struttura

A cosa serve la wildCard “?” in Java?

La **Wildcard**, rappresentata dal punto interrogativo ?, serve a superare il limite dell'**invarianza** dei Generics in Java, permettendo di scrivere codice più flessibile e riutilizzabile. Di base, sappiamo che List<Integer> *non* è un sottotipo di List<Number>. Questo ci impedirebbe di scrivere un metodo generico che accetti 'una lista di qualsiasi tipo di numero'. La Wildcard risolve questo problema introducendo la **Covarianza** o la **Controvarianza** a seconda delle necessità. Si usa principalmente in tre modi:

1. Upper Bounded Wildcard (<? extends T>) - La Covarianza Serve quando vogliamo accettare un tipo T o un qualsiasi suo **sottotipo**.

- **A cosa serve:** È utile quando dobbiamo **leggere** dati dalla struttura (agire come *Producer*).
- **Esempio:** List<? extends Number> accetta sia List<Integer> che List<Double>.
- **Vincolo:** Posso leggere i numeri (perché so che sono almeno Number), ma **non posso aggiungere** elementi (tranne null), perché non so se la lista sottostante è di Interi, Double o altro.

2. Lower Bounded Wildcard (<? super T>) - La Controvarianza Serve quando vogliamo accettare un tipo T o un qualsiasi suo **supertipo**.

- **A cosa serve:** È utile quando dobbiamo **scrivere** dati nella struttura (agire come *Consumer*).
- **Esempio:** List<? super Integer> accetta List<Integer>, List<Number> o List<Object>.
- **Vincolo:** Posso aggiungere Integer alla lista in sicurezza, perché so che la lista è fatta per contenere almeno Integer o suoi padri.

3. Unbounded Wildcard (<?>) Indica una lista di 'tipo sconosciuto'. È simile a usare List<Object>, ma più restrittivo (read-only), ed è utile quando al metodo non interessa il tipo effettivo degli elementi (es. per contare la lunghezza della lista o mescolarla).

In sintesi, per l'esame: La Wildcard serve a dire al compilatore: 'Non so esattamente che tipo ci sia qui dentro, ma prometto di rispettare questi confini'. Per ricordarsi come usarle, esiste la regola mnemonica **PECS**:

- **P**roducer **E**xtends (se devi leggere, usa extends)
- **C**onsumer **S**uper (se devi scrivere, usa super)."

Cos'è la varianza e l'invarianza in Java? Perché i Generics non sono covarianti? (8 gen 26 mattina)

la **Varianza** è una proprietà che descrive come la relazione di sottotipo tra due tipi semplici (es. Integer è sottotipo di Number) si rifletta (o meno) su tipi più complessi costruiti su di essi (come List<Integer> e List<Number>).

```
// IPOTESI ASSURDA: List<Integer> è sottotipo di List<Object>
List<Integer> listaInteri = new ArrayList<>();
listaInteri.add(10);

// Se fosse covariante, potrei fare questo:
List<Object> listaOggetti = listaInteri; // Aliasing

// Ora, tramite il riferimento "generico", inserisco un intruso:
listaOggetti.add("Sono una stringa"); // LECITO per List<Object>

// DISASTRO:
// La mia listaInteri ora contiene una Stringa!
Integer n = listaInteri.get(1); // ClassCastException a Runtime
```

In Java distinguiamo:

- **Covarianza:** Se A è sottotipo di B, allora Container<A> è considerato sottotipo di Container<B>. In Java, questo vale per gli **Array** (Integer[] è sottotipo di Number[]).
- **Invarianza:** Anche se A è sottotipo di B, non c'è **nessuna relazione** tra Container<A> e Container<B>. Sono tipi disgiunti. In Java, questo vale per i **Generics** (List<Integer> NON è sottotipo di List<Number>).

I Generics sono stati progettati come **invarianti** per garantire la **Type Safety a tempo di compilazione** e prevenire la corruzione della memoria (Heap Pollution).

Come funziona il mixin?

un **Mixin** è un meccanismo che permette di 'iniettare' (o mescolare) funzionalità aggiuntive all'interno di una classe, senza costringerla a ereditare da una specifica classe padre. Mentre l'ereditarietà classica lavora in verticale (es. Cane è *un* Animale), il Mixin lavora in **orizzontale**: fornisce una **capacità** o un **comportamento** a classi che possono essere completamente diverse tra loro (es. sia Automobile che Cane possono avere il mixin Tracciabile). Da Java 8 in poi possiamo realizzare dei mixin utilizzando le interfacce con metodi di Default.

Quali sono le caratteristiche dei Generics in Java? Qual è il loro limite?

I **Generics**, introdotti in Java 5, implementano il concetto di **polimorfismo parametrico**. Le loro caratteristiche fondamentali sono tre:

1. **Type Safety (Sicurezza dei Tipi):** Spostano il controllo dei tipi dal *runtime* (dove causerebbero crash) al *compile-time*. Il compilatore verifica che tu non stia inserendo una stringa in una lista di interi, prevenendo errori a monte.
2. **Eliminazione dei Cast Espliciti:** Non serve più fare il downcast manuale quando si estraggono dati da una collezione (es. (Integer) list.get(0)), rendendo il codice più leggibile e meno prono a errori.
3. **Riutilizzabilità del Codice:** Permettono di scrivere algoritmi generici (come l'ordinamento o la ricerca) che funzionano su qualsiasi tipo di oggetto, senza dover duplicare il codice.

Il vero limite architetturale dei Generics in Java è la Type Erasure. A differenza di C++ (che crea una copia del codice per ogni tipo usato), il compilatore Java cancella le informazioni sui tipi generici subito dopo aver controllato la correttezza del codice. Nel bytecode compilato, List<String> e List<Integer> diventano entrambi la stessa cosa: una nuda List (o List<Object>). Questo meccanismo è stato scelto per garantire la compatibilità all'indietro (Backward Compatibility) con le vecchie versioni di Java (pre-Java 5), ma comporta delle limitazioni severe.

descrivere la soluzione utilizzata all'interno della JVM per la gestione delle interfacce con itables

per capire le **itables** (Interface Tables), dobbiamo prima capire cosa non va con le classiche **vtables**.

Nell'ereditarietà singola (Classi), la JVM usa la **vTable**: se la classe Padre ha il metodo foo() all'indice 0, anche la classe Figlio avrà foo() all'indice 0. Il dispatch è velocissimo perché l'indice è fisso e noto a tempo di compilazione.

Con le **interfacce**, questo non è possibile. Immaginiamo due classi diverse, Cane e Automobile, che implementano entrambe l'interfaccia Tracciabile (metodo getPosizione()).

- In Cane, getPosizione() potrebbe essere il **5°** metodo dichiarato in memoria.
- In Automobile, lo stesso metodo potrebbe essere il **12°**.

Poiché l'indice non è costante tra le varie classi che implementano l'interfaccia, la JVM non può fare un salto diretto (vtable dispatch). Ha bisogno di una struttura di mappatura: la **itable**."

2. La Soluzione: Struttura della iTable

"Ogni classe che implementa delle interfacce possiede, oltre alla vTable, una **iTable**. Possiamo immaginarla come una lista di associazioni composta da due parti:

1. **L'intestazione dell'interfaccia:** Contiene un riferimento univoco (ID) all'interfaccia implementata.
2. **L'array degli offset:** Contiene, per ogni metodo di quell'interfaccia, l'indice corrispondente all'interno della **vTable** della classe concreta."

3. Il Funzionamento (L'istruzione invokeinterface)

"Quando la JVM esegue l'istruzione bytecode invokeinterface, il processo è leggermente più complesso rispetto a una chiamata normale:

1. **Lookup dell'Interfaccia:** La JVM scansiona la iTable dell'oggetto ricevitore cercando l'ID dell'interfaccia richiesta (es. cerca l'entry per Tracciabile).
2. **Recupero dell'Offset:** Una volta trovata l'interfaccia, guarda qual è l'indice del metodo desiderato (es. getPosizione).
3. **Salto alla vTable:** Usa quell'indice per entrare nella vTable della classe e trovare finalmente l'indirizzo del codice macchina da eseguire.

In sintesi, la iTable agisce come un **adattatore** che traduce l'indice 'astratto' dell'interfaccia nell'indice 'concreto' della classe. Le moderne JVM fanno **l'*inline caching***; JVM si ricorda l'ultima posizione trovata per una certa chiamata e la mette nella cache, in modo da accedervi prima alla chiamata successiva.

Si parli del principio di sostituzione di Liskov

Il Principio di Sostituzione di Liskov (LSP), formulato da Barbara Liskov nel 1987, definisce il concetto di Sottotipazione Comportamentale.

In parole semplici, afferma che se S è un sottotipo di T, allora oggetti di tipo T possono essere sostituiti con oggetti di tipo S senza alterare la correttezza del programma. Per garantire questo, non basta che il codice compili; devono essere rispettate tre categorie di regole: **Segnatura, Metodi e Proprietà**.

Regola della SEGNATURA→ Questa regola riguarda la definizione dei tipi nei metodi (input e output) e assicura la compatibilità delle interfacce. Si basa su due concetti chiave di varianza:

- **Controvarianza degli Argomenti (Input):** I metodi del sottotipo devono accettare gli stessi argomenti del padre o argomenti *più generali* (Supertipi).
    - *Esempio:* Se il padre accetta Cane, il figlio può accettare Animale (è più permissivo), ma non Barboncino (che sarebbe restrittivo). *Nota: Java non supporta pienamente questo aspetto nell'overriding, trattandolo come overloading*.
- **Covarianza del Risultato (Output):** I metodi del sottotipo devono restituire lo stesso tipo del padre o un tipo *più specifico* (Sottotipo).
    - *Esempio:* Se il padre restituisce Number, il figlio può restituire Integer.
- **Eccezioni:** Il sottotipo può lanciare solo le stesse eccezioni del padre o sottotipi di esse. Non può lanciare nuove eccezioni *checked* che il client del padre non si aspetta."

2. La Regola dei Metodi (Regole Semantiche)

"Questa regola riguarda il 'contratto' logico del metodo (Design by Contract).

- **Precondizioni (Cosa serve per iniziare):** Non possono essere rafforzate nel sottotipo.
    - *Spiegazione:* Il figlio non può essere più esigente del padre. Se il metodo del padre accetta qualsiasi numero positivo, il figlio non può dire 'accetto solo numeri pari'.
- **Postcondizioni (Cosa garantisco alla fine):** Non possono essere indebolite nel sottotipo.
    - *Spiegazione:* Il figlio deve garantire almeno quanto garantiva il padre. Se il padre promette di restituire un numero positivo, il figlio non può restituire un numero negativo."

Slogan per l'esame: "Il sottotipo deve **richiedere di meno** (precondizioni) e **garantire di più** (postcondizioni)."

3. La Regola delle Proprietà (Invarianti)

"Infine, questa regola riguarda lo stato dell'oggetto e la sua coerenza nel tempo.

- **Invariante di Classe:** Tutte le proprietà che devono essere sempre vere per il padre (es. velocità <= velocità_max) devono rimanere vere anche per il figlio. Il figlio non può violare le regole interne stabilite dal padre.
- **Vincolo sulla Storia (History Constraint):** Il sottotipo non può introdurre metodi che modificano lo stato in un modo che il padre non consentiva.
    - *Esempio classico:* Se definisco una classe padre **Immutabile** (come una Stringa), non posso creare un sottotipo **Mutabile**. Se un client usa il riferimento al padre pensando che non cambierà mai, e il sottotipo invece cambia, il principio è violato."

Si considerino le Classi di java riportate di seguito. Disegnare le tabelle dei metodi (dispatch vectors) delle due classi, spiegando brevemente il funzionamento e il vantaggio dello Sharing strutturale. (8 gen 26 mattina).

Nell'ambito della OOP, descrivere la differenza tra structural subtyping e nominal subtyping (8 gen 26 pomeriggio)

Il concetto chiave è: quando modifichiamo una struttura dati (ad esempio aggiungendo un elemento), non creiamo una copia completa della struttura originale. Invece, creiamo solo la parte nuova e riutilizziamo (condividiamo) i riferimenti alla parte vecchia che non è cambiata. Il vantaggio è duplice, sia in termini di Spazio che di Tempo:

1. Risparmio di Memoria: Evitiamo la duplicazione inutile di dati. Se ho una lista di un milione di elementi e ne aggiungo uno in testa, occupo solo lo spazio per un nuovo nodo, non per un milione e uno.
2. Prestazioni (Performance): Le operazioni di scrittura diventano molto veloci (spesso $O(1)$ o $O(\log n)$ per gli alberi), perché non devo scorrere e copiare tutta la struttura precedente.

Descrivere brevemente le caratteristiche e il funzionamento degli iteratori nel Java Collection frameWork. Nel seguente esempio di utilizzo, che tipo di problematica potrebbe emergere? Come si potrebbe risolvere? (8 gen 26 pomeriggio)

```
void pulisciLista(List<String> lista, String s) {
    Iterator<String> iterator = lista.iterator();

    while (iterator.hasNext()) {
        String elemento = iterator.next();
        if (elemento.equals(s)) {
            // NOTA: Questo genererà una ConcurrentModificationException!
            lista.remove(elemento);
        }
    }
}
```

Gli **Iteratori** nel Java Collection Framework sono oggetti che permettono di scorrere gli elementi di una collezione in modo sequenziale, indipendentemente dalla struttura dati sottostante (che sia una lista, un insieme o una coda).

Il loro funzionamento si basa principalmente su tre metodi definiti nell'interfaccia Iterator:

- **hasNext()**: restituisce un booleano che indica se ci sono ancora elementi da scorrere.
- **next()**: restituisce l'elemento successivo e sposta il cursore in avanti.
- **remove()**: rimuove l'ultimo elemento restituito da next() dalla collezione sottostante."

2. La problematica nel codice proposto

"Nel codice dell'esempio, la problematica principale è l'insorgere di una **ConcurrentModificationException**.

Questo accade perché gli iteratori di Java sono progettati per essere **fail-fast**: se la collezione viene modificata strutturalmente (aggiungendo o rimuovendo elementi) attraverso i metodi della classe List

(come lista.remove()) mentre l'iterazione è ancora in corso, l'iteratore rileva un'incoerenza tra il suo stato interno e quello della lista e lancia immediatamente l'eccezione per evitare comportamenti imprevedibili o corruzione dei dati."

3. Come risolvere il problema

"Esistono due modi principali per risolvere questa problematica:

1. **Usare iterator.remove() (Soluzione consigliata):** Invece di chiamare il metodo remove() sulla lista, dobbiamo chiamare quello dell'iteratore. Questo metodo è l'unico modo sicuro per modificare una collezione durante un'iterazione, poiché aggiorna correttamente lo stato interno dell'iteratore evitando l'eccezione.

Si consideri il codice Java seguente. Disegnare la tabella dei metodi (dispatch vectors) della classe Prova e la relativa iTable per la gestione delle interfacce, spiegando brevemente il funzionamento al momento della chiamata di un metodo. Cosa cambia invocando il metodo mario su un oggetto con tipo apparente Prova rispetto a farlo su un oggetto con tipo apparente I2? 10 luglio 25

```
interface I1 {
    public void pippo() ;
    public void pluto() ;
}

interface I2 {
    public void mario() ;
    public void luigi() ;
}

class Prova implements I1, I2 {
    // costruttore
    public C() { System.out.println("C") ; } // ERRORE: Il nome del costruttore (
    // metodi di I1
    public void pippo() { System.out.println("Pippo") ; }
    public void pluto() { System.out.println("Pluto") ; }
    // metodi di I2
    public void mario() { System.out.println("Mario") ; }
    public void luigi() { System.out.println("Luigi") ; }
    // altri metodi
    public void foo() {
```

Invocare su un'istanza di Prova è diretto e veloce perché l'indice nella vTable è noto a tempo di compilazione (invokevirtual). Invocare su I2 è più lento perché richiede un passaggio intermedio di lookup nella **iTable** per scoprire a quale indice della vTable corrisponde il metodo mario per quello specifico oggetto (invokeinterface). Il metodo invocato con la itable, verrebbe messo nella memoria cache, facendo inline cacheing.

Descrivere brevemente il modello della JVM (ambiente delle classi, stack e heap) e disegnare lo stato della memoria nel momento in cui il programma illustrato raggiunge il commento //ferma qui (17 dic 2024)

```
class Alpha {
    public static void main(...) { //
        Beta b1 = new Beta();
        Beta b2 = new Beta();
        b2.foo();
    }; // NOTA: questo punto e virgol
}

class Beta {
    public static int x=10;

    private int y=20;

    public void foo() {
        x+=1;
        y+=1;
        // ferma qui
    }
}
```

La memoria in Java viene divisa in 3 aree principali:

1. Ambiente delle classi→ qui si caricano le classi (Alpha, Beta in questo caso) e si memorizzano variabili statiche e nomi dei metodi
2. Stack→ Area che gestisce l'esecuzione dei metodi. Ogni volta che un metodo viene chiamato, crea un Record che contiene le variabili locali e i riferimenti. I record vengono impilati (LIFO). Nel nostro caso avremo un record per main e, sopra, un record per foo
3. Heap→Area dinamica in cui vengono allocati gli oggeti (le istanze) creati con la keyword "new". Qui ogni oggetto ha spazio per le proprie variabili d'istanza. Nel nostro caso, nell'heap gli oggetti puntati da b1 e b2 con le loro copie personali di y.

Che cosa si intende per "dynamic dispatch" nei linguaggi object oriented, e come veine realizzato in modo efficiente tramite il meccaniscmo dei dispatch vectors?

Discutere la nozione di Dynamic Dispatch in Java, illustrando la struttura del runtime

Il **Dynamic Dispatch** (o *Late Binding*) è il meccanismo con cui Java determina, **a tempo di esecuzione**, quale specifica implementazione di un metodo invocare. È la tecnologia che rende possibile il **Polimorfismo**. In Java, quando invochiamo un metodo su un oggetto, la decisione su 'quale codice eseguire' non si basa sul tipo della variabile (Tipo Apparente o Statico), ma sul tipo effettivo dell'oggetto presente in memoria (Tipo Dinamico o Concreto).

C++ e altri

Descrivi le problematiche relative all'ereditarietà multipla, e la soluzione adottata da C++.

Ecco una spiegazione strutturata per un esame, che distingue chiaramente tra i problemi di ambiguità e il problema strutturale del "Diamante", focalizzandosi sulla soluzione specifica del C++.

---

1. Le Problematiche dell'Ereditarietà Multipla

L'ereditarietà multipla si verifica quando una classe deriva direttamente da più classi base (es. class D : public B, public C). Sebbene potente, introduce due categorie di problemi.

A. Ambiguità dei Nomi (Name Collision)

Questo è il problema più banale. Se le due classi genitori (B e C) hanno un metodo con lo **stesso nome** e la stessa firma (es. foo()), e la classe figlia D non ne fa l'override, il compilatore va in confusione.

- **Situazione:** Chiamo d.foo().
- **Errore:** Il compilatore non sa se invocare B::foo() o C::foo(). Genera un errore di ambiguità.

B. Il "Diamond Problem" (Duplicazione dello Stato)

Questo è il problema più insidioso e strutturale. Si verifica quando la gerarchia forma un rombo (diamante).

- **Scenario:**
    1. C'è una classe base A (il "Nonno").
    2. B e C ereditano da A.
    3. D eredita sia da B che da C.
- **Il Problema Fisico (Memory Layout):** Senza accorgimenti, quando instanzio D, l'oggetto in memoria conterrà **due copie distinte** della classe A:
    1. Una copia "dentro" la parte B.
    2. Una copia "dentro" la parte C.
- **Conseguenze:**
    1. **Spreco di memoria:** I dati di A sono duplicati inutilmente.
    2. **Inconsistenza:** Se modifico un campo di A passando attraverso B, la modifica *non* si riflette sulla copia di A accessibile tramite C. L'oggetto è schizofrenico.
    3. **Ambiguità:** Se provo ad accedere a un campo di A da D, il compilatore non sa a quale delle due copie ti riferisci.

2. La Soluzione del C++

Il C++ offre soluzioni puntuali per entrambi i problemi.

Soluzione all'Ambiguità dei Nomi: Scope Resolution→ Per risolvere il conflitto di nomi (problema A), il programmatore deve essere esplicito usando **l'operatore di risoluzione di scope (::).**
Soluzione al Diamond Problem: Ereditarietà Virtuale→Per risolvere la duplicazione dello stato (problema B), C++ introduce l'**ereditarietà virtuale** (virtual inheritance).

Quando B e C ereditano da A, devono farlo dichiarando l'ereditarietà come virtual. Questo istruisce il compilatore a creare una sola istanza condivisa della classe base comune all'interno dell'oggetto finale, garantendo la coerenza dello stato.

Come risolve l'eredità multipla python?

Python risolve l'ereditarietà multipla linearizzando il grafo delle classi tramite l'algoritmo **C3**, producendo una lista univoca (MRO). Il problema del diamante è risolto perché la classe base comune appare una sola volta in fondo alla lista. La funzione **super()** serve a navigare dinamicamente questa lista, passando il controllo al prossimo metodo in ordine di linearizzazione.

RUST ha un garbage collector? Perché?

Che tipo di strategia adotta il linguaggio RUST rispetto alla memory safety e alla garbage collection?

Garbage Collection

Descrivere brevemente l'approccio di Garbage collection basato su "reference counting" e le problematiche di tale approccio. Usando pseudocodice o un qualunque linguaggio di programmazione, fare un esempio di porzione di programma in cui le problematiche del reference counting si manifestano. (9 gen 26)

Il reference counting utilizza un contatore messo nel decrittore del dato che un oggetto viene incrementato/decrementato ogni volta che su di un oggetto si aggiunge/rimuove un collegamento. questo approccio ha due grossi difetti:

1. **Overhead Computazionale:** Ogni singola assegnazione o passaggio di parametri richiede operazioni di scrittura in memoria per aggiornare i contatori, rallentando l'esecuzione (specialmente in ambienti multithread dove servono lock).
2. **Il Problema dei Cicli (Cyclic References):** Questa è la criticità principale. Se due o più oggetti si riferiscono a vicenda formando un anello chiusto, i loro contatori non scenderanno mai a zero, anche se il programma principale non può più raggiungerli.
    - **Risultato:** Si crea un'isola di oggetti "spazzatura" che il Reference Counting non riesce a vedere, causando un **Memory Leak**.

```java
class Nodo {
    public Nodo altro; // Riferimento a un altro nodo
}

void generaProblema() {
    // 1. Creazione: RefCount di A=1, RefCount di B=1 (puntati dalle variabili locali)
    Nodo a = new Nodo();
    Nodo b = new Nodo();

    // 2. Creazione del Ciclo:
    a.altro = b; // RefCount di B sale a 2 (variabile 'b' + campo 'a.altro')
    b.altro = a; // RefCount di A sale a 2 (variabile 'a' + campo 'b.altro')

    // 3. Rimozione dei riferimenti esterni (Root set)
    a = null; // RefCount di A scende da 2 a 1
    b = null; // RefCount di B scende da 2 a 1

    // SITUAZIONE FINALE (LEAK):
    // Il programma non può più accedere a questi oggetti.
    // Tuttavia, A punta a B (quindi B ha count=1).
    // E B punta ad A (quindi A ha count=1).
    // Poiché i contatori sono > 0, il Garbage Collector NON li eliminerà mai.
}
```

Figura 3- problema per Reference Counting

Descrivere l’approccio di garbage collection denominato copying collection. Quali sono i vantaggi e svantaggi di questo approccio?

1. Come funziona (Il meccanismo dei "Semispaces")

"La **Copying Collection** (o *Stop-and-Copy*) è un approccio radicale che risolve il problema della frammentazione della memoria dividendo l'Heap disponibile in due metà uguali, chiamate **Semispaces**:

1. **From-Space:** La metà attualmente attiva dove vengono allocati i nuovi oggetti.
2. **To-Space:** La metà attualmente vuota (di riserva).

L'algoritmo: Quando il *From-Space* si riempie, il programma si ferma (Stop-the-World). Il Garbage Collector scansiona le radici e **copia** tutti gli oggetti vivi dal *From-Space* al *To-Space*. Durante la copia, gli oggetti vengono posizionati uno attaccato all'altro (**compattazione**). Una volta terminata la copia, il *From-Space* viene considerato vuoto, i ruoli delle due metà si invertono e l'esecuzione riprende."

---

2. I Vantaggi

1. **Assenza di Frammentazione (Compattazione Implicita):** Poiché gli oggetti vivi vengono copiati nel *To-Space* in modo contiguo, non si creano mai "buchi" di memoria inutilizzati tra un oggetto e l'altro.
2. **Allocazione Velocissima (Bump Pointer):** Dato che la memoria è sempre compatta, per allocare un nuovo oggetto non serve cercare uno spazio libero in una lista (Free List). Basta incrementare un puntatore alla fine dell'area occupata (*Bump Pointer Allocation*), un'operazione che costa pochissimi cicli di clock.
3. **Efficienza con oggetti a vita breve:** Il tempo di esecuzione del GC dipende solo dal numero di oggetti **vivi**, non dalla dimensione dello heap. Se la maggior parte degli oggetti muore subito (come accade spesso), la copia è velocissima perché c'è poco da copiare.

3. Gli Svantaggi

1. **Spreco di Memoria (Dimezzamento dello Spazio):** È il difetto principale. Poiché una metà della memoria deve essere sempre tenuta vuota per la copia, l'applicazione può usare effettivamente solo il **50%** della RAM disponibile.
2. **Costo per oggetti longevi:** Se l'applicazione mantiene vivi molti oggetti per lungo tempo, il GC continuerà a copiarli avanti e indietro tra i due semispaces a ogni ciclo, sprecando CPU inutilmente.
3. **Stop-the-World:** Durante la fase di copia, l'applicazione deve essere completamente ferma. Se ci sono molti oggetti vivi da copiare, la pausa può diventare percepibile.

Discutere il ruolo e la struttura del root-set nell’algoritmo di Mark & sweep

Nell'algoritmo **Mark & Sweep**, il **Root Set** (o *GC Roots*) è fondamentale perché definisce il punto di partenza per determinare quali oggetti sono ancora in uso e quali no.

Il problema fondamentale del Garbage Collector è distinguere la 'spazzatura' dai dati utili. Non potendo chiedere al programma 'cosa ti serve?', il GC usa un criterio conservativo chiamato **Raggiungibilità (Reachability)**:

- Un oggetto è **Vivo** se è raggiungibile navigando i puntatori partendo dal **Root Set**.
- Un oggetto è **Morto** (Garbage) se non esiste alcun percorso che lo colleghi al Root Set."

2. Struttura del Root Set: Cosa contiene?

"Il Root Set non è un oggetto fisico unico nell'Heap, ma è un insieme concettuale di riferimenti che risiedono **fuori dall'Heap** e puntano **verso l'Heap**.

In un ambiente come la Java Virtual Machine (JVM), il Root Set è composto principalmente da:

1. **Stack Variables (Variabili Locali):** Sono i riferimenti presenti nello *Stack* dei thread attivi. Se un metodo è in esecuzione e ha una variabile locale x che punta a un oggetto, quell'oggetto è una radice (perché il codice lo sta usando *adesso*).
2. **Static Variables (Variabili Statiche):** I campi static delle classi caricate. Poiché le classi rimangono in memoria a lungo, qualsiasi oggetto puntato da una variabile statica è considerato vivo.
3. **Registri della CPU:** I puntatori che si trovano fisicamente nei registri del processore in quel momento.
4. **JNI References:** Riferimenti creati da codice nativo (C/C++) tramite Java Native Interface."

3. Il Root Set nell'Algoritmo (Fase di Mark)

"Durante la prima fase dell'algoritmo (**Mark**), il Garbage Collector 'ferma il mondo' (Stop-the-world) e:

1. Scansiona tutti i riferimenti presenti nel **Root Set**.
2. Marca ogni oggetto puntato direttamente da una radice come 'visitato' (o setta un bit a 1).
3. Procede ricorsivamente (solitamente con una visita in profondità - DFS) seguendo i puntatori interni di questi oggetti per marcare i figli, i nipoti, ecc.

Tutto ciò che **non** è stato toccato partendo dal Root Set alla fine di questo processo viene considerato irraggiungibile e sarà eliminato nella fase successiva (**Sweep**)

Garbage collector generazionale?

1. L'Intuizione di base: L'Ipotesi Generazionale Debole

Tutto si basa su un dato statistico osservato nella maggior parte dei programmi: **"La maggior parte degli oggetti muore giovane"**. Pochissimi oggetti sopravvivono a lungo (es. connessioni al DB, cache globali), mentre la stragrande maggioranza (es. variabili locali, iteratori, stringhe temporanee) diventa inutile quasi subito dopo essere stata creata.

2. La Strategia: "Divide et Impera"

Invece di trattare l'Heap come un unico blocco indifferenziato, il GC Generazionale lo divide in due (o più) aree principali in base all'età degli oggetti:

1. **Young Generation (Eden + Survivor Spaces):** Qui vengono creati i **nuovi oggetti**.
2. **Old Generation (Tenured):** Qui risiedono gli oggetti che sono **sopravvissuti** a vari cicli di pulizia.

3. Il Funzionamento (Il Ciclo di Vita)

1. **Allocazione:** I nuovi oggetti nascono nell'area **Eden** (Young Gen).
2. **Minor GC:** Quando l'Eden è pieno, parte una pulizia veloce (**Minor GC**).
    - Poiché la maggior parte degli oggetti è "morta", il GC li scarta immediatamente.
    - I pochi sopravvissuti vengono spostati (copiati) nei **Survivor Spaces**.
3. **Promozione (Tenuring):** Se un oggetto sopravvive a un certo numero di Minor GC (diventa "adulto"), viene promosso e spostato nella **Old Generation**.
4. **Major GC (Full GC):** Solo quando la Old Generation si riempie (evento raro), parte una pulizia profonda e costosa su tutto l'Heap (**Major GC**).

4. Il Vantaggio

L'efficienza è massimizzata: concentriamo gli sforzi frequenti (Minor GC) sulla Young Generation, dove recuperiamo tantissima memoria in pochissimo tempo (usando spesso algoritmi di *Copying*), e lasciamo in pace la Old Generation il più a lungo possibile.

Concorrenza

Discutere la problematica del deadlock nella programmazione concorrente e dire, motivando la risposta se la seguente coppia di processi paralleli descritta usando il linguaggio modello visto a lezione possa andare in deadlock o meno: (9 gen 26)

1. Teoria: Cos'è il Deadlock?

"Il **Deadlock** (o stallo) è una situazione critica nella programmazione concorrente in cui due o più processi si bloccano a vicenda, aspettando indefinitamente risorse che gli altri processi possiedono e non rilasciano.

Perché si verifichi un deadlock, devono sussistere contemporaneamente le cosiddette **4 Condizioni di Coffman**:

Il Codice dei Due Processi

Processo 1	Processo 2
`lock m1;`	`lock m1;`
`lock m2;`	`lock m2;`
`l1 := 1 + !l2;`	`lock m3;`
`lock m3;`	`l1 := !l2 + !l3;`
`l2 := !l3 + !l1;`	`unlock m3;`
`unlock m3;`	`unlock m2;`
`unlock m2;`	`l1 := !l2 + 1;`
`unlock m1;`	`unlock m1;`

1. **Mutua Esclusione:** Le risorse (es. i lock m_1, m_2) non possono essere condivise; solo un processo alla volta può usarle.
2. **Hold and Wait (Possesso e Attesa):** Un processo che possiede già una risorsa può richiederne delle altre, aspettando senza rilasciare ciò che ha già.
3. **No Preemption (Nessuna prelazione):** Le risorse non possono essere strappate via forzatamente al processo che le detiene; devono essere rilasciate volontariamente.
4. **Attesa Circolare:** Esiste una catena chiusa di processi in cui ognuno aspetta una risorsa detenuta dal successivo (es. A aspetta B, B aspetta A)."

---

2. Analisi dell'Esercizio (Immagine) – generato con AI, verifica correttamente!!

Verdetto:

"Analizzando il codice fornito nell'immagine, posso affermare con certezza che questo sistema NON può andare in deadlock."

Motivazione:

"La motivazione risiede nell'ordinamento dell'acquisizione delle risorse.

Osserviamo l'ordine in cui i lock vengono richiesti:

- **Processo 1:** Richiede, in sequenza rigorosa: lock m1 →lock m2 →lock m3.
- **Processo 2:** Richiede anch'esso, nella stessa identica sequenza: lock m1 →lock m2 → lock m3.

Affinché si verifichi un deadlock (specificamente la condizione di *Attesa Circolare*), sarebbe necessario che un processo ottenesse una risorsa (es. m_3) e ne chiedesse un'altra (es. m_1), mentre l'altro processo fa l'opposto.

In questo caso, invece, abbiamo un Ordinamento Lineare (Total Ordering) delle risorse.

Il lock m_1 agisce come un 'cancello principale':

1. Se il **Processo 1** acquisisce m_1, il **Processo 2** si blocca immediatamente alla sua prima istruzione (lock m1).
2. Il Processo 2 non può ottenere né m_2 né m_3 finché non supera lo scoglio di m_1.
3. Di conseguenza, il Processo 1 è libero di acquisire m_2 e m_3 senza competizione, completare il suo lavoro e rilasciare tutto.
4. Solo allora il Processo 2 potrà procedere.

Non essendoci possibilità di incrocio inverso (nessuno chiede m_3 prima di m_1), il ciclo non può formarsi."

Descrivere la problematica del deadlock nella programmazione concorrente, e cosa si intende per locking fine-grained e locking coarse-grained (10 luglio 2025)

1. Locking Coarse-Grained (Grana Grossa)

È l'approccio "tutto o niente". Si utilizza **un unico lock gigante** per proteggere un'intera struttura dati o un intero sottosistema.

- **Come funziona:** Se ho una Lista Condivisa, metto un lock sull'intera lista. Se il Thread A vuole leggere l'elemento 1 e il Thread B vuole scrivere l'elemento 100, B deve aspettare che A abbia finito, anche se stanno toccando dati diversi.
- **Vantaggi:**
    1. **Semplicità:** È facilissimo da implementare (es. synchronized sul metodo).
    2. **Basso Overhead:** Acquisisco e rilascio il lock una sola volta.
    3. **Sicurezza:** È molto difficile creare deadlock complessi perché c'è un solo "semaforo".
- **Svantaggi:**

1. **Bassa Concorrenza:** Crea un collo di bottiglia (bottleneck). Tutti i thread si mettono in fila indiana, trasformando l'esecuzione parallela in sequenziale.

Analogia: Chiudere a chiave il portone principale di un palazzo. Se io sono dentro, nessuno può entrare, nemmeno per andare in un appartamento diverso dal mio.

---

2. Locking Fine-Grained (Grana Fine)

È l'approccio di precisione. Si utilizzano **tanti piccoli lock**, ognuno a protezione di una piccola parte della struttura dati.

- **Come funziona:** Invece di bloccare l'intera lista, blocco solo il singolo nodo che sto modificando (es. *Lock Coupling* o *Hand-over-hand locking* nelle liste, o *Lock Striping* nelle HashMaps). Il Thread A lavora sul nodo 1, il Thread B lavora sul nodo 100 contemporaneamente.
- **Vantaggi:**
    1. **Alta Concorrenza:** Riduce drasticamente la "contesa" (contention). Più thread possono lavorare sulla stessa struttura dati contemporaneamente, purché su parti diverse.
- **Svantaggi:**
    1. **Complessità Elevata:** È difficile da scrivere correttamente.
    2. **Rischio Deadlock:** Gestendo molti lock, è facile acquisirli nell'ordine sbagliato e bloccarsi a vicenda.
    3. **Alto Overhead:** Acquisire e rilasciare 1000 lock costa tempo di CPU. Se c'è poca concorrenza, potrebbe essere addirittura più lento del coarse-grained.

Analogia: Chiudere a chiave le singole porte delle stanze. Io posso stare in cucina e tu in salotto contemporaneamente.

Perché sono necessari i meccanismi di mutua esclusione (mutex) nei linguaggi concorrenti? Che problema potrebbe verificarsi a livello assembler in un programma che esegue degli assegnamenti in parallelo? (ad es. x := x+1 | x := x+7)

I meccanismi di mutua esclusione (come i **Mutex**) sono necessari perché le operazioni su variabili condivise raramente sono **atomiche**.

Senza un Mutex che serializza l'accesso (permettendo a un solo thread alla volta di entrare nella *Sezione Critica*), ci troviamo in una situazione di **Race Condition** (Condizione di Corsa): il risultato finale del programma dipende dall'ordine imprevedibile con cui lo scheduler interfolia le istruzioni dei vari processi. Questo porta a stati inconsistenti e corruzione dei dati."

2. Il Problema a Livello Assembler (Analisi Pratica)

"Il problema nasce perché un'istruzione che ad alto livello sembra singola (es. x := x + 1), a livello macchina (Assembler/CPU) viene scomposta in **tre operazioni distinte** (ciclo *Load-Modify-Store*).

Immaginiamo di avere x = 0 inizialmente e due thread:

- **T1:** esegue x := x + 1
- **T2:** esegue x := x + 7

A livello assembler (RISC semplificato), le operazioni sono:

1. **LOAD R, x** (Carica il valore dalla RAM in un registro della CPU)
2. **ADD R, val** (Aggiungi il valore nel registro locale)
3. **STORE x, R** (Scrivi il risultato dal registro alla RAM)

Tempo	Thread T1 (`x+1`)	Thread T2 (`x+7`)	Valore in RAM (`x`)	Commento
1	`LOAD R1, x` (Legge 0)		0	T1 ha 0 nel suo registro
2		`LOAD R2, x` (Legge 0)	0	**QUI IL DANNO:** T2 legge il vecchio valore 0!
3	`ADD R1, 1` (R1 diventa 1)		0	Calcolo locale di T1
4		`ADD R2, 7` (R2 diventa 7)	0	Calcolo locale di T2
5	`STORE x, R1` (Scrive 1)		**1**	T1 salva il suo lavoro
6		`STORE x, R2` (Scrive 7)	**7**	**T2 sovrascrive T1!**

Lo scenario del "Lost Update" (Aggiornamento Perso)→ Se non usiamo i mutex, lo scheduler potrebbe interrompere T1 proprio nel mezzo di queste tre istruzioni. Vediamo un caso disastroso:

Risultato:

- **Atteso:** 8 (0 + 1 + 7)
- **Ottenuto:** 7 (oppure 1, se l'ordine di scrittura finale fosse invertito).

L'aggiornamento di T1 (+1) è andato **perso** perché T2 ha lavorato su una copia "vecchia" (stale data) della variabile, ignaro che T1 la stesse modificando."

Sintesi per l'esame

"A livello assembler, un assegnamento è una sequenza *Load-Add-Store*. Senza Mutex, è possibile che due processi leggano lo stesso valore iniziale *prima* che uno dei due abbia scritto il risultato. Questo causa il **Lost Update Problem**: l'ultimo processo che scrive sovrascrive il lavoro dell'altro, rendendo il calcolo errato."

parlare delle principali strategie per evitare i deadlock

Ci sono tre strategie principali per affrontare il problema dei deadlock:

- **Deadlock prevention**: si stabiliscono regole, controllabili staticamente (dal compilatore) sull'uso dei lock che garantiscano che i deadlock non possano verificarsi;
- **Deadlock avoidance**: l'esecuzione del programma è monitorata dal supporto a runtime del linguaggio, che si accorge di quando il programma sta per andare in deadlock e interviene cambiando l'ordine di esecuzione dei thread per evitatarlo
- **Deadlock recovery**: si permette al programma viene di entrare in deadlock, ma se ciò accade il runtime del linguaggio se ne accorge e interviene successivamente, per ripristinare uno stato senza deadlock

descrivere il 2 phase locking (2PL)

Il **Two-Phase Locking (2PL)** è un protocollo di gestione della concorrenza che garantisce la **Serializzabilità** delle transazioni (ossia, assicura che l'esecuzione concorrente produca lo stesso risultato di un'esecuzione seriale).

Il nome deriva dal fatto che ogni transazione deve rigorosamente rispettare **due fasi distinte** nel modo in cui gestisce i lock. Non può mescolare acquisizioni e rilasci a piacimento.

Le Due Fasi

Una transazione T segue il protocollo 2PL se rispetta queste fasi:

1. **Fase di Espansione (Growing Phase):**
    - La transazione può **solo acquisire** nuovi lock (o fare upgrade di lock esistenti).
    - In questa fase **non può rilasciare** nessun lock.
    - La transazione accumula tutte le risorse che le servono.
    - Il momento in cui la transazione possiede tutti i lock necessari si chiama **Locked Point**.
2. **Fase di Contrazione (Shrinking Phase):**
    - La transazione inizia a **rilasciare** i lock (o fare downgrade).
    - Da questo momento in poi, **non può più acquisire** nessun nuovo lock.
    - Una volta rilasciato il primo lock, la porta per acquistarne altri è chiusa per sempre.

La Regola d'Oro: Se una transazione rilascia un lock, non può acquisirne altri successivamente.

- **Vantaggio (Teorema del 2PL):** Se tutte le transazioni obbediscono al 2PL, lo schedule risultante è garantito essere **Serializzabile**.
- **Svantaggio:** Il 2PL **NON previene i Deadlock**. Anzi, poiché le transazioni tengono i lock in attesa di prenderne altri, il 2PL è proprio la causa principale dei deadlock nei database (che vanno risolti con timeout o grafi di attesa).