Modulo 5

Classi come Blueprint: La Natura dell'Astrazione

Una classe rappresenta un'**astrazione concettuale**. Non è un'entità concreta, ma una definizione di tipo. Quando crei una classe, stai dicendo al compilatore: "Esiste un nuovo tipo di dato nel mio programma, ed ecco come è fatto".

La classe definisce due aspetti fondamentali:

1. **Lo stato**: quali informazioni l'oggetto può contenere (attributi/campi)
2. **Il comportamento**: quali operazioni l'oggetto può compiere (metodi)

Ogni oggetto creato da quella classe è un'**istanza** separata, con il proprio spazio in memoria e i propri valori per gli attributi, ma tutti condividono la stessa struttura e gli stessi comportamenti definiti dalla classe.

Il Metodo main: Il Bootstrap dell'Applicazione

Il metodo `main` è speciale perché rappresenta il punto di contatto tra il sistema operativo e il tuo programma Java. Quando lanci un'applicazione, la JVM cerca questo metodo specifico per sapere da dove iniziare l'esecuzione.

Deve essere `static` perché la JVM non può (e non deve) creare un'istanza della classe per avviare il programma. Deve esistere un punto di ingresso che non richieda la creazione preventiva di oggetti. È il **bootstrap** dell'applicazione: tutto parte da lì, e solo dopo puoi iniziare a creare istanze di classi.

Metodi Void: Il Concetto di Side Effect

Un metodo void non restituisce un valore perché il suo scopo è produrre un **side effect** (effetto collaterale). Cosa significa? Significa che il metodo:

- Modifica lo stato interno di un oggetto
- Interagisce con il mondo esterno (stampa, scrive file, invia dati in rete)
- Altera strutture dati condivise

Il metodo void è utile quando ti interessa l'**azione** che compie, non il risultato che produce. La sua utilità sta in quello che fa, non in quello che restituisce.

Il `return;` senza valore in un metodo void serve solo per il **controllo di flusso**: ti permette di uscire dal metodo prima di arrivare alla fine naturale del blocco di codice. È uno strumento per evitare esecuzioni inutili o per gestire casi particolari.

Metodi Non-Void: Funzioni Pure e Calcoli

I metodi non-void sono più vicini al concetto matematico di **funzione**: prendono degli input (parametri) e producono un output (valore di ritorno). Sono utili quando vuoi ottenere un **risultato calcolato** che puoi poi utilizzare altrove nel tuo programma.

La differenza fondamentale con i metodi void è che i metodi non-void sono pensati per essere **usati in espressioni**: puoi assegnare il loro risultato a variabili, passarlo come argomento ad altri metodi, usarlo in condizioni, e così via.

Il Tipo di Ritorno come Contratto

Il tipo di ritorno è una forma di **contratto formale** tra il metodo e il codice che lo chiama. Quando dichiari un tipo di ritorno, stai facendo una promessa verificabile dal compilatore: "Qualunque cosa accada dentro questo metodo, alla fine restituirò un valore di questo tipo specifico".

Questo contratto è fondamentale per la **type safety** di Java: il compilatore può garantire che il valore che ricevi da un metodo sia compatibile con l'uso che ne farai. Se un metodo promette di restituire un intero, puoi tranquillamente usare quel risultato in operazioni matematiche senza controlli runtime.

Il compilatore verifica che **ogni possibile percorso di esecuzione** termini con un `return` del tipo giusto. Non basta che *probabilmente* restituirai un valore: deve essere *garantito* staticamente.

Parametri: Il Canale di Comunicazione

I parametri sono il meccanismo principale per **passare informazioni** da un contesto all'altro. Rappresentano gli input di cui un metodo ha bisogno per svolgere il suo lavoro.

Un metodo senza parametri è **autonomo**: non dipende da informazioni esterne per funzionare (anche se può accedere agli attributi dell'oggetto a cui appartiene).

I parametri creano un'interfaccia chiara: guardando la firma del metodo, sai esattamente quali informazioni devi fornire e di che tipo devono essere. Questo è parte della **documentazione implicita** del codice.

Passaggio per Valore: L'Inganno del Riferimento

Java usa **sempre** il passaggio per valore, ma questo concetto va compreso bene perché può sembrare che a volte Java passi per riferimento.

Per i Tipi Primitivi

Quando passi un tipo primitivo, il metodo riceve una **copia indipendente** del valore. È come fotocopiare un documento: puoi scrivere sulla fotocopia quanto vuoi, l'originale resta intatto.

Per gli Oggetti

Quando passi un oggetto, il metodo riceve una **copia del riferimento** (cioè dell'indirizzo di memoria). È come fotocopiare l'indirizzo di una casa: hai due copie dell'indirizzo, ma puntano comunque alla stessa casa. Se vai a quella casa e la ridipingi, il cambiamento è visibile anche usando l'altra copia dell'indirizzo.

La chiave è capire che stai copiando il **puntatore**, non l'oggetto stesso. Quindi:

- Modifiche all'oggetto puntato → visibili ovunque
- Riassegnamento del riferimento locale → nessun effetto fuori dal metodo

Questo spiega perché puoi modificare il contenuto di un array passato come parametro, ma se riassegni completamente il parametro a un nuovo array, il cambiamento non si propaga al chiamante.

Il Corpo del Metodo: Scope e Lifetime

Il corpo del metodo è uno **scope locale** dove vivono variabili temporanee. Quando il metodo termina, tutte le variabili locali (inclusi i parametri) cessano di esistere. Il loro **lifetime** è limitato all'esecuzione del metodo.

Questo è importante per capire che i parametri sono variabili locali al metodo: esistono solo durante la sua esecuzione e vengono distrutte al termine. Ecco perché modificare una variabile parametro di tipo primitivo non ha effetto fuori: stai modificando una copia che poi viene distrutta.

Approfondimento su Static: Quando qualcosa appartiene alla Classe, non all'Oggetto

Il Concetto Fondamentale di Static

Static in Java significa che qualcosa appartiene alla **classe stessa**, non alle singole istanze. È la distinzione tra "proprietà condivisa da tutti" e "proprietà individuale di ciascuno".

Quando dichiari qualcosa come static, stai dicendo: "Questo esiste una sola volta per tutta la classe, non una volta per ogni oggetto". È come la differenza tra il numero di abitanti di una città (varia per ogni città-oggetto) e il numero totale di città nel mondo (un dato unico, condiviso, che appartiene al concetto stesso di "città").

Variabili Static (Campi di Classe)

Una variabile static è condivisa da tutte le istanze della classe. Esiste **una sola copia** di quella variabile in memoria, indipendentemente da quanti oggetti crei.

Quando usarle

Le variabili static sono perfette per:

- **Contatori globali**: quante istanze sono state create di una classe
- **Configurazioni condivise**: valori che devono essere uguali per tutti gli oggetti
- **Costanti**: valori immutabili definiti una volta per tutte
- **Cache condivise**: dati che tutti gli oggetti possono consultare

Natura della condivisione

Quando modifichi una variabile static, il cambiamento è visibile a tutti. Non stai modificando "la tua copia" perché non esiste una copia per ciascuno: esiste **un'unica variabile** accessibile da qualsiasi punto del programma che possa raggiungere quella classe.

Questo crea anche un rischio: le variabili static introducono uno **stato globale condiviso**, che può rendere il codice più difficile da testare e debuggare. Se più parti del programma modificano la stessa variabile static, può diventare complesso capire chi ha cambiato cosa e quando.

Metodi Static

Un metodo static non appartiene a nessuna istanza specifica. Puoi chiamarlo direttamente sulla classe, senza bisogno di creare un oggetto.

Limitazioni fondamentali

Un metodo static **non può accedere a membri non-static** della classe. Perché? Perché i membri non-static appartengono a un'istanza specifica, e un metodo static non ha un'istanza di riferimento. Non sa a quale oggetto fare riferimento.

È come chiedere "qual è il nome del proprietario?" senza specificare di quale casa stai parlando. Se hai un metodo d'istanza, c'è un oggetto implicito (il `this`). Se hai un metodo static, non c'è nessun `this`: stai operando a livello di classe, non di oggetto.

Quando usarli

I metodi static sono ideali per:

- **Utility functions**: operazioni che non dipendono dallo stato di un oggetto (calcoli matematici, conversioni, formattazioni)
- **Factory methods**: metodi che creano e restituiscono istanze della classe
- **Operazioni sulla classe stessa**: metodi che lavorano con variabili static o che non necessitano di dati d'istanza

La natura dell'invocazione

Quando chiami un metodo static, non c'è bisogno di un oggetto ricevente. Non stai dicendo "oggetto X, esegui questa operazione". Stai dicendo "classe Y, esegui questa funzione". È un'operazione che esiste nel contesto della classe, non nel contesto di un particolare oggetto.

Il Contesto di Esecuzione: Static vs. Instance

La differenza fondamentale tra contesto static e contesto d'istanza è il **riferimento implicito**.

Nel contesto d'istanza

Quando sei in un metodo non-static, hai sempre un riferimento implicito `this`: l'oggetto su cui è stato invocato il metodo. Questo ti dà accesso a tutti i membri dell'oggetto, perché sai esattamente "quale oggetto" stai manipolando.

Nel contesto static

Quando sei in un metodo static, non c'è nessun `this`. Non stai lavorando su un oggetto specifico, stai lavorando a livello di classe. Puoi accedere solo a membri static perché quelli sono gli unici che esistono indipendentemente dalle istanze.

Il vincolo logico

Non è una limitazione arbitraria: è una conseguenza logica. Se un metodo static potesse accedere a una variabile d'istanza, quale istanza dovrebbe usare? Non c'è un modo per saperlo, perché non c'è un oggetto di riferimento.

Static e Memory: La Persistenza

Le variabili static vivono nella memoria **per tutta la durata del programma**. Non vengono create e distrutte come le variabili d'istanza (che nascono con l'oggetto e muoiono quando l'oggetto viene garbage-collected).

Implicazioni pratiche

Questo significa che:

- Le variabili static persistono tra chiamate di metodo

- Mantengono il loro valore per tutta l'esecuzione
- Possono causare memory leak se non gestite attentamente (specialmente in applicazioni server long-running)

L'area di memoria

Le variabili static risiedono in un'area speciale della memoria JVM, separata dallo heap dove vivono gli oggetti normali. Fanno parte dei metadati della classe.

CHIAMATA A METODO

La Natura della Chiamata
Una chiamata a metodo è un **trasferimento del controllo di esecuzione**. Quando chiami un metodo, il flusso del programma salta dalla posizione corrente al corpo del metodo chiamato, esegue le istruzioni lì contenute, e poi ritorna al punto di chiamata.
È come una deviazione temporanea in un percorso: lasci la strada principale, segui un sentiero secondario, e poi torni esattamente dove eri prima per continuare.

Il Meccanismo dello Stack

Ogni chiamata a metodo crea un nuovo **stack frame** (frame dello stack). Questo è uno spazio in memoria che contiene:

- I parametri passati al metodo
- Le variabili locali del metodo
- L'indirizzo di ritorno (dove tornare quando il metodo finisce)
- Informazioni di contesto per il metodo

Quando il metodo termina, il suo stack frame viene rimosso (pop dallo stack), e il controllo ritorna al chiamante. È per questo che le variabili locali di un metodo cessano di esistere quando il metodo finisce: il loro spazio di memoria viene deallocato.

La metafora della pila di piatti

Lo stack delle chiamate funziona esattamente come una pila di piatti: l'ultimo metodo chiamato è il primo che deve terminare (LIFO - Last In, First Out). Se il metodo A chiama B, e B chiama C, allora C deve finire prima di B, e B prima di A.

Chiamate su Istanze vs. Chiamate su Classi

Esistono due modalità fondamentalmente diverse di chiamare un metodo.**Chiamata su istanza (metodi non-static)**

Quando chiami un metodo non-static, lo fai **su un oggetto specifico**. C'è sempre un ricevente: l'oggetto su cui il metodo opera. Quel oggetto diventa il `this` implicito all'interno del metodo.

La sintassi è: `oggetto.nomeMetodo(parametri)`.

L'oggetto davanti al punto è cruciale: identifica quale istanza deve eseguire il metodo. Ogni istanza può avere uno stato diverso, quindi lo stesso metodo chiamato su oggetti diversi può produrre risultati diversi.**Chiamata su classe (metodi static)**

Quando chiami un metodo static, lo fai **sulla classe stessa**, non su un'istanza particolare. Non c'è un oggetto ricevente, non c'è un `this`.

La sintassi è: `NomeClasse.nomeMetodo(parametri)`.

Tecnicamente puoi chiamare un metodo static anche attraverso un'istanza, ma è considerato cattivo stile perché suggerisce falsamente che il metodo dipenda da quell'istanza. I metodi static dovrebbero sempre essere chiamati tramite il nome della classe per chiarezza.

Il Passaggio dei Parametri durante la Chiamata

Quando chiami un metodo con parametri, avviene un **processo di copia**.**Per i tipi primitivi**

Il valore della variabile viene copiato nel parametro del metodo. È una copia completamente indipendente. Qualunque cosa il metodo faccia con quel parametro non influenza la variabile originale nel chiamante.**Per gli oggetti**

Viene copiato il **riferimento** (l'indirizzo in memoria dell'oggetto). Questo significa che sia il chiamante che il metodo chiamato hanno accesso allo stesso oggetto in memoria.

Attenzione alla distinzione sottile: il riferimento stesso è copiato (quindi riassegnare il parametro non influenza il chiamante), ma entrambi i riferimenti puntano allo stesso oggetto (quindi modificare l'oggetto attraverso il parametro influenza l'oggetto originale).

L'Ordine di Valutazione

Prima che un metodo venga effettivamente chiamato, tutti i suoi argomenti vengono **valutati da sinistra a destra**.

Se scrivi qualcosa come `metodo(espressione1, espressione2, espressione3)`, Java:

1. Valuta completamente espressione1
2. Valuta completamente espressione2
3. Valuta completamente espressione3
4. Solo dopo, chiama il metodo con i valori risultanti

Questo è importante se le espressioni hanno side effect: l'ordine di esecuzione è garantito e prevedibile.

Il Valore di Ritorno

Quando un metodo non-void termina con `return valore`, quel valore viene **copiato** dal contesto del metodo al contesto del chiamante.

Sostituzione dell'espressione

La chiamata a metodo è un'**espressione** che produce un valore. Dove scrivi la chiamata, puoi immaginare che venga sostituita dal valore di ritorno.

Se scrivi `int risultato = calcolaQualcosa()`, la chiamata `calcolaQualcosa()` viene eseguita, produce un valore, e quel valore viene assegnato a `risultato`.

Uso immediato o memorizzazione

Puoi fare due cose con un valore di ritorno:

- **Usarlo immediatamente** in un'espressione senza salvarlo
- **Memorizzarlo** in una variabile per uso futuro

Entrambi gli approcci sono validi. Il primo è più conciso, il secondo può essere più leggibile e permette di riutilizzare il valore.

Chiamate Annidate

Puoi chiamare metodi all'interno di altri metodi, creando **catene di chiamate**. Ogni chiamata aggiunge un livello allo stack.

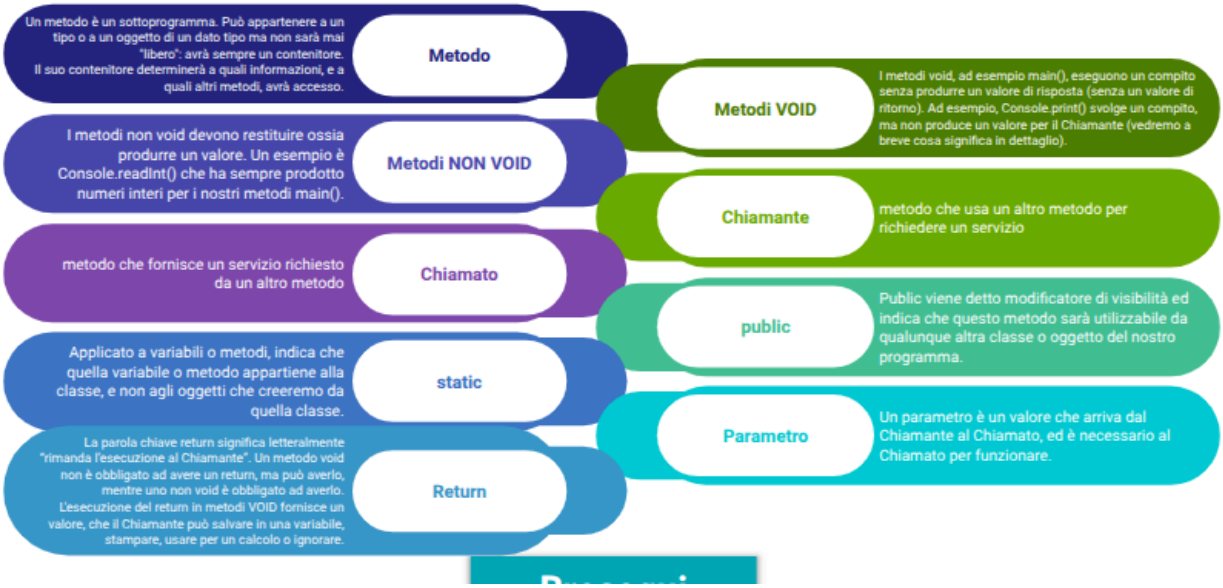
Profondità dello stack

Esiste un limite alla profondità delle chiamate annidate. Se chiami troppi metodi in sequenza (tipicamente in ricorsione infinita o eccessivamente profonda), ottieni un **StackOverflowError**: lo spazio riservato allo stack si esaurisce.

Chiamate a catena

Una chiamata a catena è quando il risultato di un metodo viene usato immediatamente per chiamare un altro metodo.

Sintassi: `oggetto.metodo1().metodo2().metodo3()`.


Metodo
Un metodo è un sottoprogramma. Può appartenere a un tipo o a un oggetto di un dato tipo ma non sarà mai "libero": avrà sempre un contenitore. Il suo contenitore determinerà a quali informazioni, e a quali altri metodi, avrà accesso.
Metodi NON VOID
I metodi non void devono restituire ossia produrre un valore. Un esempio è Console.readInt() che ha sempre prodotto numeri interi per i nostri metodi main().
Chiamato
metodo che fornisce un servizio richiesto da un altro metodo
static
Applicato a variabili o metodi, indica che quella variabile o metodo appartiene alla classe, e non agli oggetti che creeremo da quella classe.
Return
La parola chiave return significa letteralmente "rimanda l'esecuzione al Chiamante". Un metodo void non è obbligato ad avere un return, ma può averlo, mentre uno non void è obbligato ad averlo. L'esecuzione del return in metodi VOID fornisce un valore, che il Chiamante può salvare in una variabile, stampare, usare per un calcolo o ignorare.
Metodi VOID
I metodi void, ad esempio main(), eseguono un compito senza produrre un valore di risposta (senza un valore di ritorno). Ad esempio, Console.print() svolge un compito, ma non produce un valore per il Chiamante (vedremo a breve cosa significa in dettaglio).
Chiamante
metodo che usa un altro metodo per richiedere un servizio
public
Public viene detto modificatore di visibilità ed indica che questo metodo sarà utilizzabile da qualunque altra classe o oggetto del nostro programma.
Parametro
Un parametro è un valore che arriva dal Chiamante al Chiamato, ed è necessario al Chiamato per funzionare.