ASTRAZIONE - APPUNTI LIBRO

Le classi astratte come progetto concettuale

Immagina di dover disegnare il progetto di una casa. Non una casa specifica, ma l'idea generale di cosa significhi "casa": sai che dovrà avere delle stanze, delle porte, delle finestre, ma non stai ancora decidendo se sarà una villa al mare o un monolocale in città. Questo è esattamente il ruolo di una classe astratta in Java. Serve a descrivere un concetto troppo generico per esistere da solo, ma abbastanza chiaro da fare da base a qualcosa di più concreto. Proprio per questo una classe astratta non può essere istanziata direttamente: chiedere al programma di creare una "casa generica" non ha senso, devi sempre scendere di livello e dire che tipo di casa vuoi davvero.

Incompletezza intenzionale e ruolo dei metodi astratti

Dal punto di vista tecnico, quando dichiari una classe come abstract stai dicendo al compilatore e a chi leggerà il codice che quella classe è un modello incompleto. Può contenere campi, costruttori e anche metodi già implementati, ma rappresenta comunque un'idea, non un oggetto finito. Spesso questa incompletezza è resa esplicita dalla presenza di metodi astratti, cioè metodi di cui esiste solo la firma ma non il corpo. È come se nel progetto della casa fosse scritto "qui ci deve essere una cucina", senza alcuna indicazione su come sarà fatta. Il messaggio è chiaro: la cucina è obbligatoria, ma i dettagli li deciderà chi costruisce davvero la casa.

Debito implementativo e polimorfismo a runtime

Quando una sottoclasse eredita da una classe astratta, si prende sulle spalle un vero e proprio debito implementativo. Tutti i metodi astratti devono essere implementati prima o poi, altrimenti anche la sottoclasse rimarrà astratta. Questo meccanismo non è un capriccio del linguaggio, ma è ciò che rende affidabile il polimorfismo. Se io lavoro con un riferimento di tipo astratto e chiamo un metodo, so per contratto che quel metodo esiste. Quale versione verrà eseguita non lo decido io a compile time, ma il runtime tramite il late binding, scegliendo l'implementazione corretta in base al tipo reale dell'oggetto. È così che posso

scrivere codice generico, che tratta oggetti diversi nello stesso modo, lasciando al sistema il compito di invocare il comportamento giusto.

Le interfacce come contratti puri

Se le classi astratte sono come progetti generali che possono anche fornire qualche pezzo già pronto, le interfacce sono ancora più radicali. Sono contratti puri. Pensa a un ristorante che assume personale: l'interfaccia "Cuoco" dice solo che devi saper preparare antipasti, primi e secondi. Non ti fornisce ricette, utensili o indicazioni sullo stile. Puoi essere uno chef stellato o uno alle prime armi, l'importante è che tu garantisca quelle capacità. Chi usa un cuoco, cioè il client del codice, non ha bisogno di sapere come cucini, gli basta sapere che rispetti il contratto.

Separazione tra cosa e come

Tecnicamente, un'interfaccia definisce un insieme di metodi che una classe si impegna a implementare. Storicamente non conteneva né stato né implementazioni, proprio per mantenere una separazione netta tra il cosa e il come. Questo disaccoppiamento è potentissimo perché permette di scrivere codice che dipende dall'interfaccia e non dalle classi concrete, rendendo le implementazioni intercambiabili. Nel tempo Java ha arricchito le interfacce con metodi default e statici, che permettono di condividere comportamenti comuni senza rompere la compatibilità con il passato. Rimane però l'idea di fondo: l'interfaccia definisce una promessa di comportamento.

Approccio strict e approccio permissive

Questa promessa può essere mantenuta in modi molto diversi. Un esempio chiaro è il confronto tra approccio strict e approccio permissive nella gestione degli errori. Con un'implementazione strict, alla prima anomalia il sistema si ferma. È una filosofia severa, di tipo fail-fast: meglio bloccare tutto subito che andare avanti con dati potenzialmente corrotti. Questo è ideale in contesti dove l'integrità è fondamentale, come la finanza o la sanità. L'approccio permissive, invece, è più tollerante: se un dato è sbagliato lo si salta, lo si corregge con un default, si registra un avviso e si continua. Qui l'obiettivo non è la perfezione, ma la continuità operativa, soprattutto quando i dati arrivano da fonti poco affidabili.

Strategia, contesto e polimorfismo

La cosa elegante è che entrambe queste strategie possono implementare la stessa interfaccia. Il codice client non cambia, perché interagisce sempre con lo stesso contratto; a cambiare è solo la strategia scelta a runtime, incarnando perfettamente il principio del polimorfismo e lo Strategy Pattern. La decisione non è rigida ma dipende dal contesto applicativo, e proprio questa configurabilità rende il sistema più robusto e adattabile.

Implementazione multipla e composizione delle capacità

Un altro grande vantaggio delle interfacce è che una classe può implementarne quante ne vuole, superando il limite dell'ereditarietà singola delle classi. Questo permette di comporre capacità diverse senza creare gerarchie complicate. Una classe può essere confrontabile, clonabile e serializzabile allo stesso tempo, semplicemente implementando le interfacce giuste. È un modo più flessibile e pulito di costruire il comportamento degli oggetti.

Interfacce funzionali e funzioni al volo

In questo contesto si inseriscono le interfacce funzionali e le lambda. Un'interfaccia funzionale è un'interfaccia con un solo metodo astratto. Questa apparente limitazione è in realtà ciò che la rende speciale, perché consente di usare la sintassi lambda. Le lambda servono a scrivere meno codice quando hai bisogno di una funzione al volo. Invece di creare una classe o un oggetto anonimo solo per implementare un metodo, puoi scrivere direttamente il comportamento che ti serve, in modo conciso.

Lambda come sintassi compatta del polimorfismo

Dal punto di vista tecnico, una lambda è una forma compatta per creare un'istanza di un'interfaccia funzionale. I parametri e il valore di ritorno corrispondono automaticamente alla firma del metodo astratto. Dietro le quinte Java crea comunque un oggetto, ma tu non sei costretto a scrivere tutto il codice ripetitivo. Questo rende il codice più leggibile, soprattutto quando lavori con collezioni e operazioni come filtrare, trasformare o aggregare dati, dove descrivi cosa vuoi ottenere invece di come farlo passo dopo passo.

I generics come astrazione sui tipi

A completare il quadro ci sono i generics. I generics permettono di scrivere classi e interfacce che funzionano su tipi diversi senza perdere la sicurezza del controllo statico. È come progettare un contenitore che può ospitare qualunque tipo di oggetto, ma una volta deciso il tipo, il compilatore si assicura che tu non faccia errori. In questo modo eviti duplicazioni di codice e intercetti gli errori già in fase di compilazione, anche se a runtime le informazioni sui tipi vengono eliminate tramite type erasure.

Una visione unificata degli strumenti di astrazione

Mettendo insieme classi astratte, interfacce, polimorfismo, lambda e generics, si arriva al cuore della programmazione moderna in Java. Le astrazioni servono a modellare concetti generali, i contratti garantiscono comportamenti affidabili, il polimorfismo permette di cambiare implementazione senza toccare il codice client, le lambda riducono la verbosità quando serve solo un comportamento temporaneo e i generics rendono tutto riutilizzabile e sicuro. Non sono strumenti fini a se stessi, ma risposte concrete al bisogno di scrivere codice flessibile, leggibile e capace di adattarsi a contesti diversi senza rompersi.