**Annotazioni in Java**

Le annotazioni sono strumenti che aggiungono metadati al codice Java, interpretati da strumenti come il compilatore o la JVM durante l'esecuzione. Sono simili a "commenti con un significato operativo": non influenzano direttamente il comportamento del codice, ma forniscono informazioni a strumenti esterni o framework per comportamenti specifici.

**Retention delle Annotazioni**

La **Retention** indica per quanto tempo un'annotazione rimane disponibile e per chi è accessibile:

* **SOURCE**: Le annotazioni con questa retention sono visibili solo durante la compilazione e non vengono conservate nel bytecode. Sono utili per annotazioni di sviluppo che non servono a runtime.
* **CLASS**: Le annotazioni sono presenti nel bytecode, ma non disponibili a runtime. Sono visibili solo al compilatore e sono utili per informazioni interne che non devono essere lette dal codice eseguito.
* **RUNTIME**: Le annotazioni sono disponibili durante l'esecuzione e possono essere lette tramite **reflection**. Questo le rende adatte a framework e strumenti che richiedono informazioni aggiuntive durante l’esecuzione.

**Nota:** Le annotazioni di tipo RUNTIME sono importanti per chi analizza le classi con la reflection, un processo con cui si esplorano ed esaminano i dettagli di una classe durante l’esecuzione.

**Target delle Annotazioni**

Il **Target** specifica i punti all'interno del codice in cui un'annotazione può essere applicata:

* **TYPE**: L’annotazione può essere utilizzata all'inizio di una classe.
* **FIELD**: L’annotazione si applica a un attributo di classe.
* **METHOD**: L’annotazione si applica a un metodo.

In base al target, possiamo controllare dove posizionare e come utilizzare un'annotazione per definire comportamenti o fornire informazioni su specifiche parti del codice.

**Reflection e Annotazioni**

La **Reflection** è un meccanismo che consente a un programma Java di "leggere sé stesso" e accedere a informazioni sulle classi, come attributi, metodi e annotazioni. Questo è utile per framework che devono sapere in che tipo di oggetto si stanno trovando, o interpretare una classe per configurare dinamicamente un sistema.

Usando le annotazioni con **reflection**, un programma può migliorare la manutenzione e organizzazione di software complessi.

**Proxy in Java**

Un **proxy** è un componente intermedio che media la comunicazione tra due parti, spesso utilizzato per aggiungere funzionalità come sicurezza o caching senza modificare il codice principale. In Java, un proxy è implementato tramite un'interfaccia speciale chiamata **Invocation Handler**.

**Creazione di un Proxy**

Java consente di creare un proxy che implementa una specifica interfaccia. Questa configurazione permette di "avvolgere" un oggetto reale, intercettare i metodi chiamati e gestire le invocazioni tramite il metodo invoke dell’Invocation Handler.

* **Interfaccia**: Definisce i metodi che il proxy esporrà.
* **Proxy**: Implementa l'interfaccia e usa un **Invocation Handler** per gestire le chiamate ai metodi.
* **Client**: L'entità che invoca i metodi del proxy.

**Utilizzo del Proxy con MyBatis**

MyBatis utilizza i proxy per gestire le chiamate ai mapper, che rappresentano le interfacce di accesso ai dati (DAO). Quando si chiama un metodo in un mapper, MyBatis crea dinamicamente un proxy che implementa l’interfaccia e instrada la chiamata al metodo appropriato per eseguire una query SQL.

1. **Factory Class**: MyBatis utilizza una classe factory (Singleton) per aprire la connessione al database.
2. **Mapper**: Quando si chiede al framework un mapper, MyBatis fornisce un proxy che implementa l'interfaccia del mapper.
3. **Invocation Handler**: Gestisce la logica che esegue la query corrispondente al metodo chiamato.

In breve, MyBatis crea un'implementazione di Invocation Handler per far sì che, quando si richiama un metodo sull'interfaccia del mapper, venga eseguita la query SQL corrispondente. Questo meccanismo permette di interagire con il database in modo trasparente e sicuro.