Modulo 11 - Approfondimenti di programmazione a oggetti avanzata

Interfacce Funzionali

Un'interfaccia funzionale rappresenta una categoria particolare di interfacce in Java, caratterizzata dalla presenza di esattamente un metodo astratto. Questa definizione permette all'interfaccia di includere anche metodi statici, metodi di default e costanti, ma il vincolo fondamentale rimane quello di avere un solo metodo che necessita di implementazione concreta nelle classi che la realizzano. Questa caratteristica apparentemente restrittiva costituisce in realtà la base per uno dei meccanismi più eleganti del linguaggio Java moderno.

Il concetto di interfaccia funzionale diventa particolarmente rilevante quando si considera la necessità di creare implementazioni temporanee e usa-e-getta. In molti scenari di programmazione si presenta l'esigenza di fornire una implementazione rapida di un'interfaccia senza dover necessariamente creare una classe dedicata che persista nel codice. La soluzione tradizionale prevede l'utilizzo delle classi anonime, che permettono di definire contestualmente sia l'implementazione del metodo astratto sia l'istanziazione di un oggetto di quella classe senza nome. Questa tecnica, pur essendo universalmente applicabile a interfacce e classi astratte, risulta verbosa e poco elegante dal punto di vista sintattico.

Espressioni Lambda

Le espressioni lambda costituiscono una soluzione sintattica più raffinata ed elegante per gestire le interfacce funzionali. Il principio fondamentale delle lambda risiede nella loro capacità di sintetizzare in una singola istruzione tre operazioni distinte: la creazione di una classe anonima che implementa l'interfaccia funzionale, la definizione del metodo astratto richiesto e l'istanziazione di un oggetto di quella classe. La sintassi base segue il pattern input-freccia-output,

dove il compilatore Java è in grado di dedurre automaticamente il tipo dei parametri in ingresso e il tipo del valore di ritorno basandosi sulla firma del metodo astratto dell'interfaccia funzionale.

L'inferenza dei tipi rappresenta un aspetto cruciale del funzionamento delle lambda. Quando il compilatore incontra una lambda associata a una variabile di tipo interfaccia funzionale, è in grado di determinare quale sia l'unico metodo astratto da implementare senza bisogno di specificazioni esplicite. Analogamente, i parametri in ingresso vengono tipizzati automaticamente confrontando la lambda con la firma del metodo astratto atteso. Questa capacità di deduzione permette una sintassi estremamente concisa che elimina la ridondanza tipica delle classi anonime tradizionali.

Applicazioni Pratiche

L'ecosistema Java integra nativamente il concetto di interfaccia funzionale attraverso diverse implementazioni standard. L'interfaccia Predicate rappresenta un esempio paradigmatico di come le interfacce funzionali vengano utilizzate per implementare pattern comuni di programmazione. Un Predicate definisce un metodo che accetta un parametro di tipo arbitrario e restituisce un valore booleano, fungendo essenzialmente da filtro o condizione di verifica. Questa astrazione trova applicazione in numerosi contesti dove è necessario valutare se un elemento soddisfa determinati criteri.

L'integrazione delle interfacce funzionali con i metodi delle collezioni standard di Java dimostra la potenza di questo paradigma. Quando un metodo come removeIf accetta un Predicate come parametro, sta essenzialmente delegando la logica di decisione a un componente esterno fornito dal programmatore. Il metodo attraversa tutti gli elementi della collezione, invoca il metodo test del Predicate su ciascun elemento e rimuove quelli per cui il test restituisce valore vero. Questo meccanismo di callback permette di separare l'algoritmo di iterazione dalla logica di decisione, rendendo il codice più modulare e riutilizzabile.

La generalizzazione delle interfacce funzionali attraverso i generics amplifica ulteriormente la loro versatilità. Un'interfaccia come Predicate può essere parametrizzata con qualsiasi tipo, permettendo di applicare la stessa logica di filtraggio a stringhe, numeri, oggetti complessi o qualunque altra struttura dati. Questa flessibilità elimina la necessità di creare molteplici versioni specializzate

della stessa interfaccia funzionale per tipi diversi, promuovendo il riuso del codice e la composizione di comportamenti.

L'adozione delle lambda e delle interfacce funzionali rappresenta un cambiamento paradigmatico nel modo di concepire la programmazione in Java, introducendo elementi di programmazione funzionale in un linguaggio tradizionalmente orientato agli oggetti. Questo approccio ibrido permette di esprimere operazioni comuni in maniera più dichiarativa, concentrandosi sul cosa fare piuttosto che sul come farlo, pur mantenendo la compatibilità con il modello a oggetti sottostante.

Le parentesi angolari

Le parentesi angolari in Java, cioè la sintassi `<T>`, servono a esprimere il meccanismo dei generics, cioè la possibilità di scrivere classi, interfacce e metodi che non lavorano su un tipo concreto fissato in partenza, ma su uno o più tipi che verranno decisi in un secondo momento da chi userà quella classe o interfaccia. In altre parole, invece di dire "questa lista contiene sempre e solo `String`", si scrive un tipo "lista di qualcosa" e sarà il programmatore a specificare se quel "qualcosa" è `String`, `Person`, `Teacher` o altro. Le parentesi angolari racchiudono proprio questi parametri di tipo, che non sono valori o oggetti ma veri e propri tipi.

Quando si scrive `List<Person>`, si sta utilizzando l'interfaccia generica `List` indicando che, per quella particolare variabile, il tipo di elemento gestito sarà `Person`. Lo stesso meccanismo vale per `List<String>` o `List<Teacher>`. Il comportamento astratto della lista non cambia: esisteranno sempre operazioni come `add`, `remove`, `size` o l'iterazione sugli elementi. Ciò che cambia è il tipo degli oggetti contenuti, che viene deciso tramite il parametro tra parentesi angolari. Il metodo `add` accetterà esattamente il tipo scelto, ad esempio `Person` in `List<Person>`, mentre `size` continuerà a restituire un intero, indipendentemente dal tipo usato per la lista. Il concetto chiave è che la struttura dati, o l'interfaccia, viene progettata una sola volta in modo astratto, e poi concretizzata di volta in volta con tipi diversi senza riscrivere il codice.

Una classe è detta generica quando nella sua definizione compaiono uno o più parametri di tipo tra parentesi angolari, come nella forma `class NomeClasse<T>` o `class Couple<X, Y>`. In questo modo, il codice interno della classe viene scritto assumendo che esistano dei tipi astratti `X` e `Y`, senza conoscerne in anticipo la natura concreta. Nel caso di `Couple<X, Y>`, il costruttore riceve due parametri, uno di

tipo `X` e uno di tipo `Y`, e li memorizza in campi interni dello stesso tipo. Il metodo `invert` restituisce una nuova `Couple<Y, X>`, cioè una coppia con i tipi scambiati rispetto all'originale. Tutto questo avviene senza che la classe sappia se `X` e `Y` siano `String`, `Integer` o qualunque altro tipo, perché questi verranno decisi quando si dichiara e si istanzia l'oggetto, ad esempio con `Couple<String, Integer> c = new Couple<>("George", 44);`. I parametri `X` e `Y` sono quindi segnaposto per tipi reali, riempiti in fase di utilizzo.

Un dubbio tipico è perché non usare semplicemente `Object` al posto dei generics. Se si scrivesse una classe `Couple` con campi `Object x` e `Object y`, si otterrebbe una certa flessibilità, ma si perderebbe il controllo statico sui tipi. Qualunque oggetto sarebbe accettato, senza che il compilatore possa segnalare errori se si passa un tipo sbagliato. Con i generics, invece, quando si dichiara `Couple<String, Integer>` il compilatore verifica che il primo elemento sia sempre una `String` e il secondo un `Integer`, garantendo la cosiddetta type safety. Inoltre, con `Object` si sarebbe costretti a effettuare cast espliciti per tornare ai tipi originali, con il rischio di errori a runtime, mentre con i generics il tipo è noto al compilatore e non richiede cast espliciti.

I generics permettono anche di imporre vincoli sui tipi ammessi tramite le parentesi angolari. Nella sintassi `PersonCouple<X extends Person, Y extends Person>`, si dichiara una classe parametrica in cui `X` e `Y` devono essere obbligatoriamente sottotipi di `Person`. Questo significa che non si potrà scrivere `PersonCouple<String, Person>`, perché `String` non estende `Person`, mentre saranno validi tipi come `PersonCouple<Teacher, Support>` se `Teacher` e `Support` estendono `Person`. Il vantaggio è che, avendo questa garanzia, all'interno della classe si possono chiamare metodi definiti in `Person`, come `getAge()`, su `x` e `y` senza bisogno di cast e senza rischiare errori. Diventa così possibile definire un metodo come `getAgeDifference`, che calcola la differenza assoluta tra le età delle due persone. Questo genere di vincolo non sarebbe possibile con semplici campi di tipo `Object`, perché il compilatore non avrebbe alcuna certezza che esista un metodo `getAge()`.

Le parentesi angolari, quindi, non sono un mero ornamento sintattico, ma uno strumento per rendere il codice più astratto, riutilizzabile e sicuro dal punto di vista dei tipi. Permettono di scrivere strutture e comportamenti generali, come liste, insiemi, mappe o coppie, che funzionano con molteplici tipi concreti senza duplicare la logica. Allo stesso tempo, consentono di restringere i tipi accettati

quando serve accedere a metodi specifici definiti in una certa gerarchia di classi, come nel caso di `Person`. In questo quadro, interfacce come `List`, `Set`, `Map` e anche `Predicate<T>` sono tutte interfacce generiche che sfruttano le parentesi angolari per indicare il tipo di elementi su cui lavorano o l'oggetto su cui applicano un certo comportamento.