Appunti 21-01 Tecniche

Il Factory Method è un design pattern creazionale che affronta il problema della creazione di oggetti senza dover specificare esattamente quale classe concreta dovrà essere istanziata. L'essenza di questo pattern risiede nella delega del processo di istanziazione a sottoclassi o metodi specializzati, piuttosto che nell'invocazione diretta di un costruttore. Questo approccio promuove un disaccoppiamento significativo tra il codice client che necessita degli oggetti e le classi concrete che vengono effettivamente istanziate. Le factory producono le dipendenze.

Architettura del pattern

La struttura del Factory Method si fonda su quattro componenti fondamentali che interagiscono per realizzare la separazione tra interfaccia di creazione e implementazione concreta. Il Product rappresenta il tipo astratto o l'interfaccia degli oggetti che il pattern deve produrre, definendo il contratto che tutte le implementazioni concrete dovranno rispettare. Il Concrete Product costituisce l'implementazione effettiva di tale interfaccia, ovvero l'oggetto reale che viene istanziato. Il Creator dichiara il metodo factory che ha la responsabilità di ritornare oggetti di tipo Product, senza però specificare quale sottotipo concreto verrà effettivamente creato. Infine, il Concrete Creator implementa o sovrascrive il metodo factory per restituire istanze specifiche di Concrete Product.

Principi teorici e vantaggi

Il pattern si basa sul principio di ereditarietà, poiché la creazione degli oggetti viene delegata a sottoclassi che implementano il metodo factory per produrre il tipo derivato appropriato. Questo meccanismo consente di creare oggetti senza dover specificare esattamente la loro classe, permettendo così di modificare in modo flessibile il tipo di oggetto creato. La separazione tra la costruzione degli oggetti e gli oggetti stessi opera a livello della classe costruttore standard, consentendo l'applicazione dei principi di progettazione SOLID. Il pattern riduce la duplicazione di codice che altrimenti si verificherebbe se il processo di creazione

fosse incorporato direttamente nell'oggetto compositore. Inoltre, l'incapsulamento della conoscenza riguardo quale sottotipo creare svincola il framework client dai dettagli implementativi, rendendo il sistema più manutenibile e adattabile.

Contesto applicativo nel codice JDBC

Nel contesto dell'architettura JDBC, il DriverManager rappresenta il livello di gestione che opera tra l'applicazione utente e i driver di database disponibili. La classe che implementa l'interfaccia Driver può essere considerata essa stessa una factory per altri oggetti JDBC, come ad esempio oggetti di tipo Connection. Quando un oggetto Driver viene istanziato, esso si registra automaticamente nella classe DriverManager, che facilita la gestione di driver potenzialmente multipli e diversi tra loro. Il metodo statico getConnection del DriverManager agisce come un factory method che seleziona il driver opportuno tra quelli registrati e delega a tale driver la creazione della connessione appropriata al DBMS sottostante. La registrazione del driver mediante Class.forName carica dinamicamente la classe del driver, permettendo al sistema di verificare la disponibilità della libreria necessaria prima di procedere con la creazione della connessione. In questo modo, il metodo getConnection costituisce una factory che restituisce un oggetto Connection concreto senza che il codice client debba conoscere i dettagli implementativi specifici del driver utilizzato, incarnando perfettamente i principi del Factory Method pattern.

LE HASH MAP

Le HashMap sono strutture dati fondamentali in Java che associano chiavi a valori, fornendo un accesso estremamente rapido agli elementi tramite un meccanismo di hashing.

Nel contesto della ConnectionFactory, la HashMap permette di mappare il nome del database (la chiave di tipo String) alla corrispondente connessione (il valore di tipo Connection), creando un registro delle connessioni attive.

Quando si utilizza una HashMap come cache per connessioni database, il meccanismo è relativamente semplice: prima di creare una nuova connessione, si verifica se la chiave (il nome del database) è già presente nella mappa usando il metodo `containsKey` o controllando se `get` restituisce null. Se la chiave esiste, si recupera immediatamente la connessione memorizzata evitando l'overhead di

creazione, altrimenti si procede con l'istanziazione di una nuova connessione che viene successivamente inserita nella cache tramite il metodo `put`. Questo pattern è particolarmente efficiente per ridurre il carico sul database management system, poiché le connessioni vengono create solo quando strettamente necessario e riutilizzate per tutte le richieste successive.

Interfacce Funzionali e LAMBDA

Le interfacce funzionali rappresentano uno dei pilastri della programmazione funzionale in Java, introdotte formalmente con Java 8 per supportare le espressioni lambda. Un'interfaccia funzionale è definita come un'interfaccia che contiene esattamente un solo metodo astratto, indipendentemente dal numero di metodi default o static che può eventualmente includere. Questo vincolo del singolo metodo astratto è fondamentale perché permette al compilatore di comprendere univocamente quale metodo deve essere implementato quando si utilizza una lambda expression, eliminando ogni ambiguità.

> 👉🏻 L'annotazione `@FunctionalInterface` serve a dichiarare esplicitamente l'intenzione di creare un'interfaccia funzionale, e il compilatore verifica che il contratto sia rispettato generando un errore se l'interfaccia contiene più di un metodo astratto. Questa annotazione è opzionale dal punto di vista tecnico, poiché qualsiasi interfaccia con un solo metodo astratto è considerata funzionale anche senza annotazione, ma il suo utilizzo è consigliato perché rende esplicita l'intenzione del progettista e previene modifiche accidentali che violerebbero il contratto funzionale.

Le espressioni lambda costituiscono una sintassi concisa per creare istanze di interfacce funzionali senza dover definire classi concrete che le implementano.

La sintassi generale di una lambda è `(parametri) -> espressione`, dove i parametri corrispondono agli argomenti del metodo astratto dell'interfaccia funzionale e l'espressione o blocco di codice rappresenta l'implementazione del metodo. Il compilatore deduce automaticamente i tipi dei parametri dal contesto,

permettendo di omettere le dichiarazioni esplicite di tipo nella maggior parte dei casi.

> 👉🏻 Java fornisce un ricco insieme di interfacce funzionali predefinite nel package java.util.function, progettate per coprire i casi d'uso più comuni nella programmazione funzionale. Le principali includono Predicate<T> che rappresenta una condizione booleana su un input, Function<T,R> che trasforma un input di tipo T in un output di tipo R, Consumer<T> che esegue un'azione su un input senza restituire risultati, e Supplier<T> che fornisce valori senza richiedere input. Queste interfacce funzionali sono tipicamente generiche, permettendo di operare su diversi tipi di dati mantenendo la stessa logica funzionale.

Java fornisce un ricco ecosistema di interfacce funzionali nel package `java.util.function`, progettate per coprire i principali scenari della programmazione funzionale. Queste interfacce rappresentano concetti astratti come funzioni, azioni, predicati e fornitori, e costituiscono i target type per lambda expressions e method references.

Le quattro interfacce funzionali principali

Predicate<T> rappresenta una funzione booleana che accetta un singolo argomento e restituisce un valore boolean. Il metodo astratto è `boolean test(T t)`, utilizzato principalmente per operazioni di filtraggio e verifiche condizionali. Ad esempio, `Predicate<Item> pFI = (item) -> item.getPrice() > 500` crea un predicato che testa se il prezzo di un item supera 500, perfetto per essere usato con `stream().filter()`. L'interfaccia fornisce anche metodi default come `and()`, `or()` e `negate()` per comporre predicati complessi.

Consumer<T> rappresenta un'operazione che accetta un singolo argomento e non restituisce alcun risultato. Il metodo funzionale è `void accept(T t)`, utilizzato tipicamente per operazioni con side-effects come stampe, logging o

aggiornamenti di database. Un esempio classico è `Consumer<Integer> consumer = x -> System.out.println(x)` che viene invocato con `consumer.accept(5)`. I Consumer sono frequentemente utilizzati con il metodo `forEach()` delle collezioni: `items.forEach(item -> System.out.println(item.getItemName()))`. L'interfaccia offre il metodo default `andThen()` per concatenare consumer sequenzialmente.

Function<T,R> rappresenta una funzione che accetta un argomento di tipo T e produce un risultato di tipo R. Il metodo funzionale è `R apply(T t)`, consentendo trasformazioni di dati da un tipo ad un altro. Questa interfaccia è fondamentale nelle operazioni di mapping degli Stream, dove si trasforma ogni elemento della collezione in un altro tipo o valore. Function fornisce metodi di composizione come `compose()` e `andThen()` che permettono di concatenare funzioni creando pipeline di trasformazioni.

Supplier<T> rappresenta una funzione che non accetta argomenti ma produce un risultato di tipo T. Il metodo funzionale è `T get()`, utilizzato per fornire valori su richiesta senza richiedere input. I Supplier sono ideali per lazy initialization, generazione di valori default, o factory methods: `Supplier<String> getConnectionUrl = () -> "jdbc://localhost:5432/users"`. Esistono varianti primitive come `IntSupplier`, `DoubleSupplier` e `LongSupplier` per evitare il boxing dei tipi primitivi.

Varianti per due argomenti

Java fornisce versioni "Bi" delle principali interfacce funzionali che accettano due parametri invece di uno. **BiConsumer<T,U>** accetta due argomenti tramite `void accept(T t, U u)` ed è utilizzato frequentemente con il metodo `forEach()` delle mappe: `map.forEach((key, value) -> System.out.println("Key:" + key + " Value:" + value))`. **BiFunction<T,U,R>** accetta due argomenti e restituisce un risultato tramite `R apply(T t, U u)`, permettendo operazioni che combinano due valori: `BiFunction<Integer, Integer, String> biFunction = (num1, num2) -> "Result:" + (num1 + num2)`. **BiPredicate<T,U>** verifica condizioni su due argomenti restituendo un boolean tramite `boolean test(T t, U u)`: `BiPredicate<Integer, String> condition = (i, s) -> i > 20 && s.startsWith("R")`.

L’uso congiunto di lambda expressions e method references rappresenta un’evoluzione sintattica del linguaggio che non cambia la semantica di fondo, ma rende il codice più espressivo e dichiarativo.

A livello teorico, tutto ruota intorno al concetto di “obiettivo funzionale”: ogni lambda o method reference è un’implementazione compatta di un’interfaccia funzionale.

Nel caso della lambda usata dentro `removeIf`, l’idea astratta è quella di definire un predicato, ossia una funzione booleana che decide se un elemento deve essere eliminato oppure no. L’operatore logico `||` non è specifico delle lambda: rappresenta semplicemente la composizione di due condizioni booleane in OR. Il fatto che sia scritto dentro la lambda significa solo che la “regola di eliminazione” è descritta come un’espressione logica: per ogni elemento della collezione, la funzione associata al predicato viene invocata, valuta le due condizioni e restituisce vero se almeno una è soddisfatta. A livello concettuale, quindi, `removeIf` non sa nulla di prezzi o descrizioni: sa solo che possiede una funzione booleana da chiamare su ogni elemento, e delega alla lambda la decisione finale.

Per l’ordinamento, il Comparator espresso con `(a,b) -> a.title.compareTo(b.title)` incarna l’idea astratta di “relazione d’ordine” tra coppie di oggetti. Un Comparator, in termini teorici, definisce una funzione che prende due elementi e restituisce un segnale numerico che esprime la loro posizione relativa: negativo se il primo precede il secondo, positivo se lo segue, zero se sono equivalenti in quell’ordinamento. Il fatto che, internamente, si usi `compareTo` delle stringhe è solo una scelta concreta per costruire questa relazione sfruttando l’ordinamento naturale dei titoli. L’algoritmo di sort non conosce i dettagli del confronto: si limita a chiamare ripetutamente la funzione di confronto fornita dal Comparator e a riorganizzare gli elementi in modo coerente con i valori restituiti.

Infine, il method reference come `Console::print` è la forma più essenziale del concetto di “passare un comportamento” già esistente. Dal punto di vista teorico, una lambda che inoltra semplicemente il proprio parametro a un metodo è ridondante: l’informazione rilevante è “chiama quel metodo con l’argomento che ti viene dato”.

Il method reference raccoglie proprio questa informazione nel modo più compatto possibile, trattando un metodo come un valore che può essere passato in giro ed eseguito più tardi. Le diverse forme teoriche di method reference corrispondono a diversi modi di riferirsi a un comportamento: un metodo statico visto come funzione pura, un metodo di istanza collegato a un oggetto specifico, un metodo d'istanza applicabile a qualunque oggetto di un certo tipo, o un costruttore interpretato come funzione che crea nuove istanze. In tutti i casi, però, l'idea di fondo è la stessa: separare la definizione del comportamento dal momento in cui viene invocato, così da poterlo combinare e riutilizzare facilmente in contesti diversi, come `forEach`, `sort` o altre operazioni sulle collezioni.

Method reference operator ::

L'operatore `::` in Java è chiamato "method reference operator" ed è stato introdotto in Java 8 come sintassi alternativa compatta per esprimere lambda expressions che si limitano a invocare un metodo esistente. A livello teorico, questo operatore rappresenta il modo in cui Java permette di trattare i metodi come valori di prima classe, ossia come entità che possono essere passate come argomenti, restituite da funzioni o assegnate a variabili, pur mantenendo il sistema dei tipi basato su interfacce funzionali.

un operatore di riferimento a metodo che crea un'istanza di un'interfaccia funzionale puntando a un metodo compatibile. Quando scrivi `ClassName::methodName`, stai costruendo un oggetto che implementa un'interfaccia funzionale il cui metodo astratto corrisponde alla firma del metodo referenziato. Il compilatore si occupa di creare il collegamento tra l'interfaccia funzionale richiesta dal contesto e il metodo referenziato, verificando che i parametri e il tipo di ritorno siano compatibili.

Quattro forme teoriche

Esistono quattro forme concettuali di method reference che corrispondono a modi diversi di intendere un metodo come funzione astratta.

La prima forma è il **reference a metodo statico**, espressa come `ClassName::staticMethodName`. In questo caso, il metodo è completamente slegato da qualsiasi istanza e può essere visto come una funzione pura che accetta parametri e restituisce un risultato. Ad esempio, `StringUtils::capitalize` rappresenta la

funzione che, dato un oggetto String, restituisce una nuova String capitalizzata, senza bisogno di alcun contesto aggiuntivo.

La seconda forma è il **reference a metodo di istanza di un oggetto specifico**, espressa come `object::instanceMethod`. Qui il metodo è legato a un'istanza particolare già esistente, quindi il reference cattura sia il comportamento del metodo sia lo stato dell'oggetto su cui verrà invocato. Questa forma è utile quando hai già un oggetto e vuoi usare uno dei suoi metodi come callback o handler.

La terza forma, teoricamente più sottile, è il **reference a metodo di istanza di tipo arbitrario**, espressa come `ClassName::instanceMethod`. Nonostante la sintassi sia identica a quella dei metodi statici, la semantica è completamente diversa: il reference non specifica quale istanza verrà usata, ma indica che il primo parametro della funzione rappresentata sarà l'oggetto su cui invocare il metodo. Ad esempio, `String::compareTo` produce una funzione che accetta due stringhe e invoca `compareTo` della prima sulla seconda, quindi il primo parametro diventa implicitamente il receiver del metodo.

La quarta forma è il **reference a costruttore**, espressa come `ClassName::new`. Questa forma tratta il costruttore come una funzione factory che, dati i parametri appropriati, produce una nuova istanza della classe. È particolarmente utile con gli Stream quando si vuole trasformare una collezione di valori in una collezione di oggetti.