11. Programmazione Funzionale in Java: Lambda Expressions e Stream API

Questo capitolo esplora le **Lambda Expressions** e la **Stream API**, due delle più significative aggiunte a Java 8 che hanno introdotto un forte supporto alla programmazione funzionale. Queste funzionalità permettono di scrivere codice più conciso, leggibile e parallellizzabile.

5.1. Introduzione alle Lambda Expressions

Le Lambda Expressions (o espressioni lambda) rappresentano un **modo conciso per definire funzioni anonime** (senza nome) che possono essere passate come argomenti a metodi o memorizzate in variabili.

5.1.1. Le Novità di Java 8

Java 8, rilasciato nell'estate del 2014, ha introdotto le lambda come una delle principali novità, portando lo stile di programmazione funzionale nel linguaggio. Questo ha permesso di scrivere codice più compatto, chiaro ed elegante in determinate situazioni, impattando vari aspetti del linguaggio e delle librerie.

5.1.2. Motivazioni per le Lambda Expressions

Prima di Java 8, per passare un comportamento (una funzione) come argomento a un metodo, era necessario creare una classe anonima (o una classe interna). Questo portava a codice verboso e difficile da leggere.

Esempio (pre-Java 8): Ordinamento di una lista con classe anonima

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class OldStyleSorting {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Ordinamento in base alla lunghezza della stringa usando una classe anonima
        Collections.sort(names, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        });

        System.out.println(names); // Output: [Bob, Alice, Charlie]
    }
}
```

Questo codice, pur funzionante, è prolisso per un'operazione semplice come la definizione di un criterio di confronto.

5.1.3. Sintassi delle Lambda Expressions

Una lambda expression ha tre parti principali:

1. **Lista dei parametri**: Tra parentesi, come in un metodo. I tipi possono essere omessi se il compilatore può inferirli.
2. **Freccia (`->`)**: Separa i parametri dal corpo della lambda.
3. **Corpo della lambda**: Può essere una singola espressione (il cui risultato è il valore di ritorno) o un blocco di istruzioni tra parentesi graffe.

Sintassi Generale: `(parametri) -> { corpo della lambda }`

Esempi di sintassi:

- **Zero parametri**: `() -> System.out.println("Hello")`
- **Un parametro (tipo inferito)**: `s -> s.length()`
- **Un parametro (tipo esplicito)**: `(String s) -> s.length()`
- **Più parametri**: `(a, b) -> a + b`
- **Blocco di codice**: `(int x, int y) -> { int sum = x + y; return sum; }`

5.1.4. Interfacce Funzionali

Le lambda expressions non sono tipi a sé stanti in Java. Possono essere usate solo in contesti in cui è attesa un'**interfaccia funzionale**. Un'interfaccia funzionale è un'interfaccia che contiene **un solo metodo astratto** (Single Abstract Method - SAM) → metodo che non ha un'implementazione.

- Le espressioni lambda forniscono un modo conciso per implementare un metodo di un'interfaccia funzionale.

L'annotazione `@FunctionalInterface` è opzionale ma raccomandata per indicare che un'interfaccia è destinata a essere funzionale. Il compilatore verificherà che contenga un solo metodo astratto.

Perché usare i metodi astratti?

I metodi astratti sono fondamentali per l'**astrazione** e il **polimorfismo** nella programmazione orientata agli oggetti:

- **Definire un "contratto"**: Un metodo astratto agisce come un "contratto". Dice: "Ogni sottoclasse che eredita da questa classe astratta DEVE avere questo metodo e DEVE implementarlo." Questo garantisce che tutte le sottoclassi abbiano un comportamento specifico, anche se il modo in cui quel comportamento viene implementato varia da sottoclasse a sottoclasse.
- **Implementazione variabile**: Permette di definire un comportamento comune a un livello di astrazione superiore, ma lascia i dettagli dell'implementazione alle singole sottoclassi. Ad esempio, una classe `Animale` potrebbe avere un metodo astratto `faiSuono()`. Ogni animale (cane, gatto, leone) fa un suono, ma il suono specifico è diverso, quindi ogni sottoclasse implementerà `faiSuono()` in modo diverso (`"Bau"`, `"Miao"`, `"Roar"`).
- **Evitare l'istanza di classi incomplete**: Poiché una classe astratta può avere metodi senza implementazione, non avrebbe senso crearne un'istanza direttamente. Dichiararla `abstract` impedisce la creazione di oggetti di quel tipo incompleto, forzando gli sviluppatori a creare istanze di sottoclassi concrete che hanno fornito tutte le implementazioni necessarie.
- **Gerarchie di classi**: Sono spesso usati in gerarchie di classi per stabilire un framework o uno scheletro comune, delegando le specificità alle classi figlie.

Esempio di Interfaccia Funzionale:

```
@FunctionalInterface
interface MyFunction {
    int apply(int a, int b);
}
```

Utilizzo di Lambda con Interfacce Funzionali:

```
public class UseLambda {
    public static void main(String[] args) {
        // Implementazione di MyFunction tramite lambda
        MyFunction adder = (a, b) -> a + b;
        System.out.println("Somma: " + adder.apply(5, 3)); // Output: Somma: 8

        // Esempio di ordinamento con lambda (Java 8+)
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
```

```
        names.add("Charlie");

        // Comparator è un'interfaccia funzionale (metodo compare)
        Collections.sort(names, (s1, s2) -> s1.length() - s2.length());
        System.out.println(names); // Output: [Bob, Alice, Charlie]
    }
}
```

Java fornisce numerose interfacce funzionali predefinite nel package `java.util.function`, tra cui:

Interfaccia Funzionale	Metodo Astratto	Descrizione
`Predicate<T>`	`boolean test(T t)`	Accetta un argomento e restituisce un booleano. Usato per i filtri.
`Consumer<T>`	`void accept(T t)`	Accetta un argomento e non restituisce nulla. Usato per le azioni.
`Function<T, R>`	`R apply(T t)`	Accetta un argomento di tipo `T` e restituisce un risultato di tipo `R`. Usato per le trasformazioni.
`Supplier<T>`	`T get()`	Non accetta argomenti e restituisce un risultato di tipo `T`. Usato per la generazione di valori.
`UnaryOperator<T>`	`T apply(T t)`	Accetta un argomento di tipo `T` e restituisce un risultato di tipo `T` (specializzazione di `Function`).
`BinaryOperator<T>`	`T apply(T t1, T t2)`	Accetta due argomenti di tipo `T` e restituisce un risultato di tipo `T` (specializzazione di `BiFunction`).

5.1.5. Method References (Riferimenti a Metodi)

I riferimenti a metodi sono una forma ancora più concisa di espressioni lambda, utilizzabili quando una lambda si limita a chiamare un metodo già esistente.

Sintassi:

- `Object::instanceMethod` (riferimento a un metodo d'istanza di un oggetto specifico)
- `Class::staticMethod` (riferimento a un metodo statico di una classe)
- `Class::instanceMethod` (riferimento a un metodo d'istanza di un oggetto arbitrario di un tipo specifico)
- `Class::new` (riferimento a un costruttore)

Esempi:

- `System.out::println` (equivalente a `s -> System.out.println(s)`)
- `String::valueOf` (equivalente a `i -> String.valueOf(i)`)
- `Person::getIncome` (equivalente a `p -> p.getIncome()`)
- `ArrayList::new` (riferimento al costruttore di `ArrayList`)

5.2. Stream API

La Stream API, introdotta in Java 8 insieme alle lambda, fornisce un modo potente e flessibile per elaborare sequenze di dati in modo dichiarativo.

5.2.1. Il Concetto di Stream

- **Flusso di Dati**: Uno `Stream` rappresenta un **flusso sequenziale** (anche infinito) di **dati omogenei**.
- **Uso Singolo**: Uno stream può essere consumato una sola volta. Dopo essere stato attraversato, **non può essere riutilizzato**.
- **Dichiarativo**: Gli stream sono più dichiarativi degli iteratori. Non indicano passo-passo come l'informazione viene processata, ma cosa deve essere fatto.
- **Lazy Evaluation (Valutazione Pigra)**: Ove possibile, uno stream manipola i suoi elementi in modo "lazy". I dati vengono processati mano a mano che servono, non sono memorizzati tutti assieme come nelle `Collection`. Questo può migliorare le prestazioni per grandi dataset o stream infiniti.
- **Catene di Trasformazioni**: È possibile creare "catene di trasformazioni" di stream (implementate con decorazioni successive) in modo funzionale, per ottenere computazioni non banali dei loro elementi, con codice più compatto e

leggibile.

- **Parallelizzazione**: Gli stream possono essere eseguiti in parallelo (parallel streams), sfruttando i core multipli della CPU per migliorare le prestazioni su operazioni computazionalmente intense.

USO SINGOLO: Perché questa limitazione?

Questa caratteristica fondamentale degli Stream in Java è intrinsecamente legata al loro **design e alla loro filosofia**, che mirano all'efficienza e alla parallelizzazione. Ecco i motivi principali:

1. Efficienza e Risorse

Uno dei motivi principali è l'**efficienza**. Gli Stream sono progettati per elaborare dati in modo **lazy** (pigro) e **on-demand**. Quando crei uno stream, i dati non vengono immediatamente elaborati o copiati in una nuova struttura. L'elaborazione avviene solo quando viene invocata un'operazione terminale.

Per esempio, se hai una lista di milioni di elementi e applichi diverse operazioni intermedie (`filter`, `map`, ecc.), lo stream non elabora completamente ogni operazione intermedia su tutti gli elementi. Invece, processa un elemento alla volta attraverso l'intera pipeline di operazioni intermedie, fino all'operazione terminale. Una volta che un elemento è stato processato e "passato" attraverso lo stream, **non c'è bisogno di conservarlo in memoria** per un potenziale riutilizzo. Questo riduce notevolmente l'uso della memoria e le risorse computazionali.

Se gli stream fossero riutilizzabili, il sistema dovrebbe mantenere lo stato di tutti gli elementi e di tutte le operazioni intermedie, il che vanificherebbe il concetto di elaborazione "lazy" e potrebbe portare a un consumo di memoria e CPU insostenibile per grandi dataset.

2. Evitare Effetti Collaterali (Side Effects)

Il design degli Stream incoraggia uno **stile di programmazione funzionale**, in cui le operazioni non producono "effetti collaterali" visibili o modificano lo stato esterno. Consentire il riutilizzo di uno stream potrebbe indurre gli sviluppatori a dipendere da uno stato interno che potrebbe cambiare tra un'esecuzione e l'altra, portando a comportamenti imprevedibili e difficili da debuggare.

Un stream "usa e getta" rafforza l'idea che ogni operazione su di esso è una trasformazione pura che porta a un risultato finale, senza lasciare dietro di sé uno stato riutilizzabile che possa essere inquinato.

3. Supporto alla Parallelizzazione

Gli Stream sono stati progettati pensando alla **parallelizzazione**. Quando un'operazione stream viene eseguita in parallelo, i dati possono essere suddivisi e processati su core diversi. Il fatto che un elemento venga elaborato una sola volta e poi "dimenticato" facilita enormemente la gestione della concorrenza e previene problemi di sincronizzazione che sorgerebbero se gli stessi elementi dovessero essere rielaborati da thread diversi.

5.2.2. Differenze tra `Collection` e `Stream`

Caratteristica	`Collection`	`Stream`
Natura	Struttura dati in memoria (contiene dati)	Flusso di dati (elabora dati)
Modificabilità	Modificabile	Immutabile (le operazioni producono nuovi stream)
Iterazione	Può essere iterata più volte	Può essere consumata una sola volta
Valutazione	Eager (tutti gli elementi sono in memoria)	Lazy (elementi processati su richiesta)
Operazioni	Operazioni su elementi individuali	Operazioni di alto livello (filtro, mappa, riduzione)
Parallelismo	Non intrinsecamente parallelo	Facilmente parallelizzabile (`.parallelStream()`)

5.2.3. Pipeline di Stream: Operazioni Intermedie e Terminali

Le operazioni su uno stream formano una "pipeline". Una pipeline di stream è composta da:

1. **Sorgente**: Da dove provengono i dati (es. `Collection`, array, `Stream.of()`, `Files.lines()`, `Random.ints()`).
2. **Zero o più operazioni intermedie**: Trasformano lo stream in un altro stream. Sono "lazy" e non eseguono l'elaborazione finché non viene invocata un'operazione terminale. Possono essere concatenate.
   - Esempi: `filter()`, `map()`, `flatMap()`, `distinct()`, `sorted()`, `peek()`, `limit()`, `skip()`.
3. **Una singola operazione terminale**: Produce un risultato o un effetto collaterale, e chiude lo stream. È "eager" e avvia l'elaborazione della pipeline.

- Esempi: `forEach()`, `collect()`, `reduce()`, `count()`, `min()`, `max()`, `sum()`, `average()`, `anyMatch()`, `allMatch()`, `noneMatch()`, `findFirst()`, `findAny()`.

5.2.4. Esempi di Operazioni su Stream

Consideriamo una lista di oggetti `Person` (con nome, città, reddito, lavori) per dimostrare le operazioni di stream.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.Set;
import java.util.Collections;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.LongStream;
import java.util.Random;

// Classe Person (definita nel PDF 15-lambdas_slides.pdf, Pagina 36)
class Person {
    private final String name;
    private final Optional<String> city;
    private final double income;
    private final Set<String> jobs;

    public Person(String name, String city, double income, String... jobs) {
        this.name = name;
        this.city = Optional.ofNullable(city);
        this.income = income;
        this.jobs = Set.of(jobs); // Immutable Set
    }

    public String getName() { return name; }
    public Optional<String> getCity() { return city; }
    public double getIncome() { return income; }
    public Set<String> getJobs() { return Collections.unmodifiableSet(jobs); } // Copia difensiva

    @Override
    public String toString() {
        return "Person [name=" + name + ", city=" + city.orElse("N/A") + ", income=" + income + ", jobs=" + jobs + "]";
    }
}

public class UseStreamsOnPerson {
    public static void main(String[] args) {
        final List<Person> list = new ArrayList<>();
        list.add(new Person("Mario", "Cesena", 20000, "Teacher"));
        list.add(new Person("Rino", "Forlì", 50000, "Professor"));
        list.add(new Person("Lino", "Cesena", 110000, "Professor", "Dean"));
        list.add(new Person("Ugo", "Cesena", 20000, "Secretary"));
        list.add(new Person("Marco", null, 4000, "Contractor"));

        // Esempio 1: Calcolare il reddito totale delle persone di Cesena
        // Filtra le persone con città presente e uguale a "Cesena"
        // Mappa ogni persona al suo reddito (double)
        // Somma tutti i redditi
        final double result = list.stream()
            .filter(Person::getCity().isPresent()) // Operazione intermedia: filtra le persone con città definita
            .filter(p -> p.getCity().get().equals("Cesena")) // Operazione intermedia: filtra per città "Cesena"
```

```
    .mapToDouble(Person::getIncome) // Operazione intermedia: trasforma in uno stream di double
    .sum(); // Operazione terminale: somma gli elementi
System.out.println("Reddito totale delle persone di Cesena: " + result);
* *Provenienza*: 15-lambdas_slides.pdf, Pagina 38

// Esempio 2: Trovare i lavori distinti delle persone di Cesena
// Filtra le persone di Cesena
// flatMap: trasforma ogni Set<String> di lavori in un unico stream di String
// distinct: rimuove i duplicati
// collect: raccoglie i risultati in una stringa unita
final String jobsInCesena = list.stream()
    .filter(p -> p.getCity().filter(x -> x.equals("Cesena")).isPresent())
    .flatMap(p -> p.getJobs().stream()) // Trasforma un Set di lavori in un flusso di lavori
    .distinct() // Rimuove i lavori duplicati
    .collect(Collectors.joining(" | ", "[[", "]]")); // Raccoglie in una stringa
System.out.println("Lavori distinti a Cesena: " + jobsInCesena);
* *Provenienza*: 16-stream_slides.pdf, Pagina 29

// Esempio 3: Reddito medio dei professori
// Filtra le persone che sono "Professor"
// Mappa al reddito
// Calcola la media (OptionalDouble per gestire il caso di stream vuoto)
final OptionalDouble avg = list.stream()
    .filter(p -> p.getJobs().contains("Professor"))
    .mapToDouble(Person::getIncome)
    .average(); // Operazione terminale: calcola la media
System.out.println("Reddito medio dei professori: " + avg.orElse(0.0));
* *Provenienza*: 16-stream_slides.pdf, Pagina 29

// Esempio 4: Generazione di numeri primi (stream infinito)
// iterate: genera uno stream infinito di numeri interi a partire da 2
// filter: mantiene solo i numeri primi (verifica se non è divisibile per nessun numero precedente)
// limit: prende solo i primi 1000 numeri
// mapToObj: converte Long in String
// collect: unisce in una stringa
System.out.println("\nPrimi 1000 numeri primi:");
System.out.println(
    LongStream.iterate(2, x -> x + 1) // Sorgente: stream infinito di numeri da 2 in poi
        .filter((i) -> LongStream.range(2, (long) Math.sqrt(i) + 1).noneMatch(j -> i % j == 0)) // Filtro: è primo?
        .limit(100) // Limita ai primi 100 numeri primi
        .mapToObj(String::valueOf) // Trasforma in stringa
        .collect(Collectors.joining(",", "[", "]")) // Raccoglie in una stringa
);
* *Provenienza*: 16-stream_slides.pdf, Pagina 31 (adattato per chiarezza e prestazioni)

// Esempio 5: Simulazione di lanci di dadi
// range: genera numeri da 0 a 9999
// map: simula il lancio di due dadi (r.nextInt(6)+r.nextInt(6)+2)
// boxed: converte int in Integer (per poter usare collect)
// collect: raggruppa per valore e conta le occorrenze
final Random r = new Random();
System.out.println("\nFrequenza di somma di due dadi su 10000 lanci:");
System.out.println(
    IntStream.range(0, 10000) // Sorgente: stream di 10000 numeri (indici)
        .map(i -> r.nextInt(6) + r.nextInt(6) + 2) // Trasforma: somma di due dadi
        .boxed() // Converte int in Integer
        .collect(Collectors.groupingBy(i -> i, Collectors.counting())) // Raggruppa e conta
);
```

```
     * *Provenienza*: 16-stream_slides.pdf, Pagina 31 (adattato per chiarezza)
   }
}
```

5.2.5. Tipi di Stream Primitivi

Per evitare l'overhead di boxing/unboxing quando si lavora con tipi primitivi (`int`, `long`, `double`), Java fornisce stream specializzati:

- `IntStream`
- `LongStream`
- `DoubleStream`

Questi stream offrono metodi specifici per operazioni numeriche come `sum()`, `average()`, `min()`, `max()`. Possono essere convertiti in stream di oggetti (`Stream<Integer>`, `Stream<Long>`, `Stream<Double>`) tramite il metodo `boxed()`.

5.2.6. `Optional<T>`: Gestione dei Valori Presenti/Assenti

`Optional<T>` è una classe container introdotta in Java 8 per rappresentare un valore che potrebbe essere presente o assente. È un modo per evitare `NullPointerException` e per rendere più esplicito il fatto che un valore potrebbe non esserci.

Metodi chiave:

- `Optional.of(T value)`: Crea un `Optional` con il valore specificato (lancia `NullPointerException` se `value` è `null`).
- `Optional.ofNullable(T value)`: Crea un `Optional` con il valore specificato, o un `Optional` vuoto se `value` è `null`.
- `isPresent()`: Restituisce `true` se il valore è presente.
- `isEmpty()`: Restituisce `true` se il valore è assente (introdotto in Java 11).
- `get()`: Restituisce il valore se presente, altrimenti lancia `NoSuchElementException`. **Usare con cautela dopo aver verificato `isPresent()`.**
- `orElse(T other)`: Restituisce il valore se presente, altrimenti restituisce un valore di default.
- `orElseGet(Supplier<? extends T> supplier)`: Restituisce il valore se presente, altrimenti invoca il `Supplier` per ottenere un valore di default.
- `orElseThrow()`: Restituisce il valore se presente, altrimenti lancia `NoSuchElementException`.
- `orElseThrow(Supplier<? extends X> exceptionSupplier)`: Restituisce il valore se presente, altrimenti invoca il `Supplier` per lanciare un'eccezione personalizzata.
- `ifPresent(Consumer<? super T> consumer)`: Esegue l'azione specificata se il valore è presente.
- `filter(Predicate<? super T> predicate)`: Se un valore è presente e soddisfa il predicato, restituisce un `Optional` contenente il valore, altrimenti un `Optional` vuoto.
- `map(Function<? super T, ? extends R> mapper)`: Se un valore è presente, applica la funzione di mappatura e restituisce un `Optional` contenente il risultato.
- `flatMap(Function<? super T, ? extends Optional<? extends R>> mapper)`: Simile a `map`, ma la funzione di mappatura restituisce già un `Optional`.

Esempio:

```
import java.util.Optional;

public class UseOptional {
    public static void main(String[] args) {
        Optional<String> name = Optional.of("Alice");
        Optional<String> emptyName = Optional.ofNullable(null);

        System.out.println("Nome presente? " + name.isPresent()); // true
        System.out.println("Nome vuoto? " + emptyName.isEmpty()); // true

        name.ifPresent(s -> System.out.println("Ciao, " + s)); // Ciao, Alice
```

```
        String user = emptyName.orElse("Guest");
        System.out.println("Utente: " + user); // Utente: Guest

        // Usando map e filter
        Optional<Integer> length = name.map(String::length);
        System.out.println("Lunghezza del nome: " + length.orElse(0)); // 5

        Optional<String> longName = name.filter(s -> s.length() > 6);
        System.out.println("Nome lungo presente? " + longName.isPresent()); // false
    }
}
```

Questo conclude il Capitolo 5, che ha coperto le Lambda Expressions e la Stream API, due concetti chiave per la programmazione funzionale in Java. Ho cercato di spiegare ogni aspetto in modo chiaro e dettagliato, fornendo esempi pratici e integrando le informazioni con le ricerche sul web.

Sono pronto per procedere con l'ultimo capitolo quando lo desideri!