# CORSO JAVA AESYS
## Parte Prima

2022

www.aesystech.it

# Collections Framework e Stream API parte seconda

- Le interfacce Set e SortedSetFramework Collections
- Le interfacce Map e SortedMap
- Introduzione alla libreria Stream
- Definizioni di Stream e pipeline
- Classi Optional
- Reduction Operations

# Le interfacce Set e SortedSet

L'interfaccia Set, (in italiano "insieme" nel senso di insieme matematico) rappresenta una collezione non ordinata di elementi in cui non sono ammessi duplicati. In particolare l'interfaccia Set non definisce altri metodi rispetto a Collection, bensì impone regole più restrittive quando questi vengono utilizzati. Questo significa che se per esempio aggiungiamo due volte tramite il metodo add() lo stesso elemento, questo verrà aggiunto solamente una volta.
L'interfaccia SortedSet, invece rappresenta una collezione ordinata di elementi in cui non sono ammessi duplicati. Questa definisce alcuni metodi per lavorare su range di elementi o sugli estremi della collezione.
Un'implementazione di Set è HashSet, mentre un'implementazione di
SortedSet è TreeSet. HashSet è più performante rispetto a TreeSet ma non gestisce l'ordinamento. Entrambe queste classi non ammettono elementi duplicati e sono di sicuro le più utilizzate implementazioni di insiemi. HashSet garantisce prestazioni migliori e dovrebbe essere sempre preferita a TreeSet, a meno che non si voglia iterare in maniera ordinata sugli elementi della collezione:

```
Set<String> set = new TreeSet<>();
set.add("c"); set.add("a"); set.add("b");
set.add("b"); set.forEach(System.out::println);
```

L'output sarà: a,b,c

# Le interfacce Map e SortedMap

Una mappa è un oggetto che mappa chiavi a valori, e che non può contenere chiavi duplicate. È definita usando due tipi parametro, una per le chiavi (K, inziale di “key” che in inglese significa “chiave”) e una per i valori (V, iniziale di “value” che in inglese significa “valore”):

```
public interface Map<K,V> { //...
```

Seguono i metodi fondamentali definiti da questa interfaccia:

- V put(K key, V value) aggiunge una coppia chiave-valore alla mappa e ritorna il vecchio valore associato alla chiave (se era presente), oppure null.
- V get(Object key) restituisce il valore associato alla chiave passata in input. Può tornare null se la chiave non esiste (oppure se l’implementazione della mappa supporta valori nulli e alla chiave è associato un valore null).
- V remove(Object key) rimuove il mapping chiave-valore associato alla chiave specificata e ritorna il vecchio valore associato alla chiave (se era presente), oppure null.
- boolean containsKey(Object key) restituisce true o false a seconda se la chiave specificata è presente
- boolean containsValue(Object value) restituisce true o false a seconda se il valore specificato è presente.
- Set<K> keySet() ritorna un Set contenente tutte le chiavi presenti nella mappa.

# Introduzione alla libreria Stream API 1/2

Uno stream rappresenta un flusso che ha come sorgente una collezione di dati, e su cui si possono eseguire operazioni complicate con uno sforzo di programmazione minimo, spesso basato su espressioni lambda e reference a metodi. Tipicamente uno stream viene creato a partire da una collection tramite la chiamata al metodo stream(), ma ci sono diversi modi per ottenere oggetti Stream. Per esempio è possibile istanziare uno Stream passando un array al metodo statico of:

```
Stream <String> stringsStream = Stream.of(arrayDiStringhe);
```

oppure passando all'altra versione del metodo of() direttamente dei valori:

```
Stream <String> stringsStream = Stream.of("Take", "The", "Time");
```

Inoltre, un'oggetto Stream è molto utile per iterare sulle collezioni, mediante il suo metodo forEach(), e da questo sfruttare le cosiddette operazioni di aggregazione. Queste ultime vengono utilizzate congiuntamente a espressioni lambda e reference a metodi, rendendo il codice estremamente potente e allo stesso tempo conciso. Queste espressioni infatti, ritornano sempre un altro oggetto di tipo Stream, fornendo quindi la possibilità di concatenare le chiamate a queste funzioni. Queste concatenazioni di chiamate sono definite comunemente pipeline. Per esempio, considerando l'oggetto smartphones dei paragrafi precedenti, con il seguente statement:

```
smartphones.stream().filter(s->"Samsung".equals(s.getMarca())).   forEach(s->System.out.println(s));
```

stamperemo: Samsung Note

# Introduzione alla libreria Stream API 2/2

Infatti, abbiamo chiamato il metodo stream() sulla collection smartphones, ottenendo appunto un primo oggetto Stream. Su di esso abbiamo invocato il metodo filter() che prende in input un Predicate (che, ricordiamo, era usato proprio per implementare dei test che ritornano un booleano. Quindi gli abbiamo passato un'espressione lambda che implementa un controllo con il metodo equals() sulla marca dell'elemento. Il metodo filter() ritorna nuovamente un'oggetto Stream relativo alla collezione filtrata, che conterrà un unico elemento (visto che nella collezione c'è un unico smartphone di marca Samsung). Su questo stream invochiamo il metodo foreach() della classe Stream, che esegue un'iterazione completa sugli elementi della collection, e permette di eseguire un'operazione su ogni elemento della stessa. Anche in questo caso ci limitiamo a stampare gli elementi.

L'operazione filter() viene detta "operazione di aggregazione" perché progettata per restituire un nuovo stream, su cui invocare altre operazioni.

# Definizioni di Stream e pipeline

Solitamente usiamo le collection per raggruppare elementi in un solo oggetto, per poi eseguire operazioni sui loro elementi all'interno di cicli. Uno stream (che in italiano possiamo tradurre come flusso) è una sequenza di elementi su cui possiamo iterare. L'iterazione è unica e una volta terminata, lo stream non è più utilizzabile. A differenza di una collection, uno stream non immagazzina elementi, ma permette di operare su questi elementi attraverso pipeline. Questo ha il grande vantaggio di consentirci di eseguire tali operazioni in modo molto conciso ed efficiente. Il vantaggio più evidente è quando concateniamo le operazioni da fare sugli elementi della collezione tramite pipeline. Possiamo definire pipeline (che in italiano possiamo tradurre come conduttura inteso come "tubo composto da diversi tubi messi in sequenza per far passare un flusso di qualcosa") una sequenza di operazioni di aggregazione. Una pipeline è costituita da tre element:

- una **sorgente** (in inglese **source**): potrebbe essere una collection o un array.
- zero o più **operazioni di aggregazione** (oppure **operazioni intermedie**): queste operazioni hanno la caratteristica di ritornare un nuovo oggetto Stream.
- una **operazione terminale:** un metodo che restituisce un risultato che non è un oggetto Stream

# Classi Optional

OptionalDouble è una delle importanti classi di Java dette classi optional. Le classi optional sono delle classi wrapper che consentono di evitare NullPointerException. Queste infatti possono immagazzinare valori di un certo tipo, e sfruttare i propri metodi affinché l'eccezione più famosa non sia lanciata, senza dover ricorrere ai soliti controlli di nullità. Oltre ad OptionalDouble, esistono le classi OptionalInt, OptionalLong e Optional<T>. Per capire come funzionano tali classi supponiamo di avere il seguente metodo:

```
public static String getTitoloMaiuscolo(String titolo) {     if (titolo != null) {
return titolo.toUpperCase();
   }
   return "NESSUN TITOLO"; }
```

Questo metodo restituisce la stringa parametro titolo in maiuscolo, oppure la stringa NESSUN TITOLO nel caso la stringa titolo, sia null. Per evitare di incorrere in una NullPointerException, abbiamo dovuto fare dei verbosi controlli di nullità all'interno del nostro metodo. Per evitare la verbosità possiamo utilizzare un oggetto Optional ridefinendo il metodo in questo modo:

```
public static String getTitoloMaiuscoloOpt(String titolo) {
   Optional<String> opt = Optional.ofNullable(titolo);    return opt.orElse("NESSUN
TITOLO").toUpperCase(); }
```

Il metodo ofNullable() restituisce un oggetto Optional che contiene l'oggetto titolo. Se titolo è null l'oggetto Optional sarà comunque creato con un contenuto null.

# Reduction Operations

Tornando agli stream, abbiamo già asserito che il metodo average() è considerato una reduction operation (in italiano una operazione di riduzione) ovvero un metodo che a partire dagli elementi di uno stream, ritorna un solo valore. Altre tipiche implementazioni di operazioni di riduzione sono metodi come max(), min(), sum() e count() i cui nomi sono auto esplicativi. Per esempio con il seguente statement:

```
long count = smartphones.stream().   filter(s -> s.getMarca().equals("Apple")).count();
```

Esistono però anche altre operazioni di riduzione che ritornano collezioni invece che singoli elementi. Stiamo parlando dei metodi reduce() e collect().

- reduce()
- Collect()
- metodi summarizing