Java Stream

Java Stream

- Java fornisce un nuovo package aggiuntivo in Java 8 utilizzabile importando **java.util.stream**.
- Questo package è composto da classi, interfacce ed enum per consentire operazioni in stile ***funzionale*** sugli elementi.
- Utilizzando ***stream***, è possibile elaborare i dati in modo **dichiarativo** simile alle istruzioni SQL, in luogo di un approccio procedurale, basato su loop.
- Utilizzando le collection in Java, uno sviluppatore deve utilizzare loop ed eseguire controlli ripetuti mentre, con Stream, le operazioni sono codificate a livello dichiarativo.
- Con gli Stream si sfruttano le capacita delle **CPU multiprocessore** effettuando operazioni di estrazione, modifica, sort etc, in parallelo

Stream Caratteristiche Generali

- **Creazione**

  Ci sono molti modi per creare una istanza di uno stream.

  Una volta creato, lo stream **non modificherà il suo input**, permettendo invece la creazione di istanze multiple da un unico source

- **Sequenza di elementi**

  Uno stream fornisce un insieme di elementi di un tipo specifico in modo sequenziale. Uno stream ottiene/calcola elementi su richiesta.

  Non memorizza mai gli elementi.

- **Source**

  Gli stream prendono in input Collections, Arrays, oppure risorse di I/O, etc

- **Operazioni aggregate**

  Stream supporta operazioni aggregate come filtro, mappatura, limitazione, riduzione, ricerca, corrispondenza e così via.

Stream Caratteristiche

❑ Pipelining

La maggior parte delle operazioni di stream restituisce lo stream stesso in modo che il loro risultato possa essere pipelined (impilato).
Queste operazioni sono chiamate operazioni **intermedie** e la loro funzione è quella di prendere input, elaborarli e restituire l'output al target.
Il metodo **collect()** è un'operazione **terminale** che è normalmente presente alla fine dell'operazione di pipelining *per contrassegnare la fine dello Stream*.

❑ Automatic iterations

Le operazioni dello Stream eseguono le iterazioni internamente sugli elementi di origine forniti, a differenza delle Collections in cui è richiesta un'iterazione esplicita.

Stream features

- **Stream non memorizza elementi**

  Trasferisce semplicemente elementi da una fonte come una struttura dati, un array o un canale I/O, attraverso una pipeline di operazioni computazionali.

- **Stream è funzionale per natura**

  Le operazioni eseguite su uno stream non ne modificano l'origine. Ad esempio, il filtro di un flusso ottenuto da una collection produce un nuovo stream senza gli elementi filtrati, invece di rimuovere elementi dalla collection di origine.

- **Stream è Lazy**

  Il codice è valutato solo quando richiesto

- **Stream permette un solo utilizzo**

  Gli elementi dello stream sono visitati solo una volta durante la vita dello stream. Come un iteratore, un nuovo stream deve essere rigenerato per rivisitare gli stessi elementi.

Stream Source

- Uno Stream rappresenta una **sequenza di oggetti** da diverse sorgenti, che supporta operazioni di **aggregazione**.

- Gli Stream possono essere generati a partire da:
  - ***String***
  - ***Collection***
  - **Array**
  - **Primitives**
  - **File**
  - ***Stream.builder()***
  - ***Stream.generate()***
  - ***Stream.iterate()***

Stream Empty

- In caso di creazione di uno stream vuoto si dovrebbe usare il metodo ***empty()***
- Si usa spesso il metodo *empty()* al momento della creazione, per evitare di restituire stream *null* quando non ci sono elementi

```
Stream<String> streamEmpty = Stream.empty();
```

Stream Of String

- Gli elementi dello stream generato da una Stringa possono essere le singole linee, se la stringa contiene dei fine linea (/n, CR x'0D')

```
System.out.println("13.1 Stream per estrazione linee da String");
str = "Questo\n e\n un\n corso di Java";
listFromStream = str.lines()
                    .collect(Collectors.toList());
System.out.println(listFromStream);
```

```
13.1 Stream per estrazione linee da String
[Questo,  e,  un,  corso di Java]
```

Stream Of String

- Gli elementi dello stream generato da una Stringa possono essere i singoli caratteri che la compongono

```
//13 Stream per estrazione Chars da String
System.out.println();
System.out.println("13 Stream per estrazione Chars da String");
str = "Questo è un corso di Java";
str.chars().forEach(o->System.out.println(o));
```

```
13 Stream per estrazione Chars da String
81
117
101
115
```

Stream Of String

- Il metodo ***chars()*** di String produce uno stream modellato dalla classe ***IntStream,*** poiché non c'è nessuna interfaccia *CharStream* in JDK

```
IntStream streamOfChars = "abc".chars();
```

- Esempio di suddivisione di una stringa in sottostringhe in accordo alla *RegEx* specificata

```
Stream<String> streamOfString =
  Pattern.compile(", ").splitAsStream("a, b, c");
```

Stream Of Collection

- Si può creare uno stream da qualsiasi tipo di Collection (Collection, List, Set)

```
Collection<String> collection = Arrays.asList("a", "b", "c");
Stream<String> streamOfCollection = collection.stream();
```

Stream Of Array

- Uno stream può essere generato a partire da un array

```
Stream<String> streamOfArray = Stream.of("a", "b", "c");
```

- Oppure da un array esistente o da una sua parte

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> streamOfArrayFull = Arrays.stream(arr);
Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);
```

Stream Of Primitives

- Possono essere creati stream a partire da tre tipi primitivi: ***int***, ***long*** e ***double***.
- Poiché *Stream<T>* è una interfaccia generica e non c'è modo di usare dati primitivi come tipi parametro, sono state create tre nuove interfacce speciali
  - ***IntStream***
  - ***LongStream***
  - ***DoubleStream***

```
IntStream intStream = IntStream.range(1, 3);
LongStream longStream = LongStream.rangeClosed(1, 3);
```

Stream Of Primitives

- ***range(int startInclusive, int endExclusive)***
  Crea uno stream ordinato dal primo al secondo parametro e il risultato **non** include il secondo parametro, che rappresenta l'upper bound
- ***rangeClosed(int startInclusive, int endInclusive)***
  Si comporta come il caso precedente ma il secondo parametro è icluso.
  Sin da Java 8 la classe Random fornisce un ampio range di metodi per generare stream di primitivi.
  Per esempio il codice seguente crea un *DoubleStream* che ha tre elementi

```
Random random = new Random();
DoubleStream doubleStream = random.doubles(3);
```

Stream Of File

- La classe java NIO ***Files*** permette di generare uno ***Stream<String>*** di un file di testo attraverso il metodo ***lines()***
- Ogni linea del file diventa un elemento dello stream
- Il ***Charset*** può esere specificato come argomento del metodo *lines()*

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =
  Files.lines(path, Charset.forName("UTF-8"));
```

Stream With Builder

- Si utilizza ***Stream.Builder()*** per generare uno stream inserendo i singoli elementi, che possono essere quindi inseriti dinamicamente
- Il tipo desiderato dovrebbe essere addizionalmente specificato nella parte destra dello statement, altrimenti il metodo ***build()*** creerà una istanza di ***Stream<Object>***

```
Stream<String> streamBuilder =
  Stream.<String>builder().add("a").add("b").add("c").build();
```

Stream With Generate

- Il metodo ***generate()*** accetta un ***Supplier<T>*** per la generazione dell'elemento. Poiché lo stream risultante è infinito, si dovrebbe specifcare la lunghezza desiderata, altrimenti il metodo ***generate()*** lavorerà fino al raggiungimento dei limiti di memoria

- Crea una sequenza di 10 stringhe con il valore «element»

```
Stream<String> streamGenerated =
  Stream.generate(() -> "element").limit(10);
```

Stream With Iterate

- Il metodo ***iterate()*** fornisce un altro modo per la creazione di uno stream infinito

```
Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
```

- Il primo elemento dello stream risultante è il primo parametro del metodo *iterate()*
- Alla creazione di ogni successivo elemento, la funzione Lambda viene applicata all'elemento precedente
- Nell'esempio precedente il secondo elemento sarà 42

Stream References

- Si può istanziare uno Stream e ottenere un riferimento per accedervi, purchè siano richiamate solo operazioni elementari. L'esecuzione di operazioni terminali rende lo stream inaccessibile
- Per dimostrare ciò costruiamo un esempio di operazioni elementari su uno stream, che potrebbero ovviamente essere concatenate

```
Stream<String> stream =
  Stream.of("a", "b", "c").filter(element -> element.contains("b"));
Optional<String> anyElement = stream.findAny();
```

Stream References

- Il tentativo di riutilizzare lo stesso riferimento dopo aver eseguito l'operazione terminale `stream.findAny()` provocherà *IllegalStateException*

```
Optional<String> firstElement = stream.findFirst();
```

- Poiché *IllegalStateException* è una *RuntimeException* il compilatore non può segnalare il problema
- Quindi è bene ricordare che i **Java 8 stream non possono essere riusati**

Stream References

- Per rendere il codice precedente a prova di exception si dovrebbe scrivere:

```
List<String> elements =
  Stream.of("a", "b", "c").filter(element -> element.contains("b"))
    .collect(Collectors.toList());
Optional<String> anyElement = elements.stream().findAny();
Optional<String> firstElement = elements.stream().findFirst();
```

Stream Pipeline 1

- Per eseguire una serie di operazioni sugli elementi di uno stream e aggregare i risultati, sono necessarie tre parti:
  - **source**
  - **intermediate operation(s)**
  - **terminal operation**
- Le operazioni intermedie restituiscono un nuovo **stream** modificato
- Per esempio, per creare un nuovo stream da uno esistente eliminando alcuni elementi, si utilizzerà il metodo ***skip()***

```
Stream<String> onceModifiedStream =
    Stream.of("abcd", "bbcd", "cbcd").skip(1);
```

Stream Pipeline 2

- Se si desiderano più modifiche da apportare a uno stream, si possono **concatenare** più operazioni **intermedie**
- Per esempio si vuole sostituire ogni elemento del corrente *Stream<String>* con una sub-string dei primi caratteri
- Questa operazione può essere fatta concatenando i metodi ***skip()*** e ***map()***

```
Stream<String> twiceModifiedStream =
  stream.skip(1).map(element -> element.substring(0, 3));
```

Stream Pipeline 3

- Il metodo ***map()*** ha come parametro di input una Lambda expression, il cui scopo è estrarre i primi 3 caratteri di ogni elemento con cui creare il nuovo stream
- Lo stream in se è privo di valore; si è interessati al risultato di operazioni terminali, che possono essere un valore di qualche tipo oppure una azione applicata ad ogni elemento dello stream
- Il modo più conveniente di utilizzare gli stream è con una **stream pipeline** ovvero una catena di stream source, operazioni intermedie e una terminale

```
List<String> list = Arrays.asList("abc1", "abc2", "abc3");
long size = list.stream()
                .skip(1)
                .map(element -> element.substring(0, 3))
                .sorted()
                .count();
```

Stream Reduction 1

- Il meccanismo di riduzione permette di ottenere un output **ridotto**, dello **stesso tipo** degli elementi dello stream sorgente
- Stream ha molte operazioni terminali di aggregazione (riduzione): *count(), max(), min(), e sum()* che operano tutte secondo implementazioni predefinite
- Esiste la necessità di avere specifici meccanismi di riduzione codificati dall'utente
- Esistono due metodi che permettono di effettuare una riduzione
  - ***reduce()***
  - ***collect()***

Stream Reduction 2

- ***reduce()***

  Ci sono tre variazioni di questo metodo, che differiscono per la firma e il tipo restituito

  - **identity**

    Il valore iniziale di un accumulatore o il valore iniziale di uno stream è empty e non c'è nulla da accumulare.

  - **accumulator**

    Una funzione che specifica la logica di aggregazione degli elementi. Poiché l'accumulatore crea un nuovo valore per ogni step di riduzione, la quantità di nuovi valori eguaglia le dimensioni dello stream e soltanto l'ultimo valore è utile, cosa non buona dal punto di vista performance

Stream Reduction 3

- *reduce()*
  - **combiner**

    Una funzione che aggrega il risultato dell'accumulatore.

    Si utilizza combiner solo in modo parallelo per ridurre i risultati di accumulatori da thread differenti

Stream Reduction 4

```
OptionalInt reduced =
IntStream.range(1,4).reduce((a, b) -> a + b);
```

***reduced* = 6 (1 + 2 + 3)**

```
int reducedTwoParams =
IntStream.range(1,4).reduce(10, (a, b) -> a + b);
```

reducedTwoParams* = *16 (10 + 1 + 2 + 3)

Stream Reduction 5

```
int reducedParams = Stream
        .of(1, 2, 3)
        .reduce(10, (a, b) -> a + b, (a, b) -> {
            log.info("combiner was called");
             return a + b;
           });
```

- Il risultato sarà lo stesso come nel precedente esempio (**16**) e il combiner NON sarà chiamato. Per funzionare lo stream dovrebbe essere parallelo

Stream Reduction 5

```
int reducedParallel =
Arrays.asList(1,2,3).parallelStream()
             .reduce(10, (a, b) -> a + b, (a, b) -> {
                log.info("combiner was called");
                return a + b;
});
```

- Il risultato è ora differente (**36**) e il combiner è stato chiamato due volte.
  Qui la riduzione lavora con il seguente algoritmo: l'accumulatore esegue 3 volte ogni elemento dello stream a *identity.* Queste azioni sono fatte in parallelo.
  Come risultato abbiamo **(10 + 1 = 11; 10 + 2 = 12; 10 + 3 = 13;)**
  Ora il combiner può mergiare questi 3 risultati e necessita di due iterazioni per ottenere **(12 + 13 = 25; 25 + 11 = 36)**

Stream Collect() Method 1

- La riduzione di uno stream può essere eseguita anche da un'altra operazione terminale, attraverso ile metodo ***collect()***
- Il metodo ***collect()*** accetta un argomento di tipo ***Collector***, che specifica il meccanismo di riduzione
- Sono già disponibili innumerevoli collectors predefiniti per le operazioni più comuni, accedibili attraverso il ***Collectors* type**
- Consideriamo la lista seguente

```
List<Product> productList = Arrays.asList(
new Product(23, "potatoes"),
new Product(14, "orange"),
new Product(13, "lemon"),
new Product(23, "bread"),
new Product(13, "sugar"));
```

Stream Collect() Method 2

- **Convertiamo uno stream in una collection (Collection, List, Set)** a partire dalla List precedente `productList`

```
List<String> collectorCollection = productList
                                   .stream()
                                   .map(Product::getName)
                                   .collect(Collectors.toList());
```

Stream Collect() Method 3

- **Riduzione a String**

```
String listToString = productList
        .stream()
        .map(Product::getName)
        .collect(Collectors.joining(", ", "[", "]"));
```

- Il metodo *joiner()* può avere da uno a tre parametri (**Delimiter**, **Prefix**, **Suffix**)
- La cosa più conveniente del metodo *joiner()* è che non c'è necessità di verificare se lo stream raggiunge la sua fine per applicare il suffisso e per non applicare il delimiter. Il *Collector* gestisce tutto questo.

Stream Collect() Method 4

- Elaborazione valore medio di tutti gli elementi dello stream

```
double averagePrice = productList.stream()
.collect(Collectors.averagingInt(Product::getPrice))
;
```

Stream Collect() Method 5

- **Elaborazione somma di tutti gli elementi numerici dello stream**

```
int summingPrice = productList.stream()
.collect(Collectors.summingInt(Product::getPrice));
```

- I metodi *averagingXX()*, *summingXX()*, *summarizingXX()*, possono lavorare con dati primitivi (*int*, *long*, *double*) e le loro classi wrapper (*Integer*, *Long*, *Double*).
- Una feature potente di questi metodi è fornire il mapping e come risultato non è necessario codificare una *map()* addizionale prima del metodo *collect()*

Stream Collect() Method 6

- **Produzione informazioni statistiche sugli elementi dello stream**

```
IntSummaryStatistics statistics =productList.stream()
.collect(Collectors.summarizingInt(Product::getPrice)
);
```

- Attraverso l'istanza risultante di tipo *IntSummaryStatistics* si possono creare report statistici applicando il metodo *toString()*
- Il risultato sarà una mnormale stringa del tipo *"IntSummaryStatistics{count=5, sum=86, min=13, average=17,200000, max=23}."*
- Risulta anche semplice estrarre da questo oggetto valori separate per *count*, *sum*, *min*, e *average* con I metodi *getCount()*, *getSum()*, *getMin()*, *getAverage()*, and *getMax()*.

Stream Collect() Method 7

- **Raggruppamento elementi dello stream in base a specifica funzione**

```
Map<Integer, List<Product>> collectorMapOfLists =
productList.stream()
.collect(Collectors.groupingBy(Product::getPrice));
```

  - Lo stream è stato ridotto a una *Map*, che raggruppa tutti i prodotti per il loro prezzo

Stream Collect() Method 7

- Divisione elementi dello stream in gruppi in accordo a qualche predicato

```
Map<Boolean,List<Product>>mapPartioned=productList
    .stream().collect(Collectors.partitioningBy(
                element->element.getPrice() > 15));
```

Stream Collect() Method 8

❑ **Collector supplementare per trasformazioni addizionali**

```
Set<Product> unmodifiableSet = productList.stream()
      .collect(Collectors.collectingAndThen(
Collectors.toSet(),Collections::unmodifiableSet));
```

- In questo caso particolare il collector ha convertito un stream in un *Set* dopodichè ha creato un *Set* non modificabile a partire da questo

Stream Collect() Method 8

- **Collector custom**

  Il metodo più semplice per creare un collector personalizzato è quello di usare il metodo ***of()*** del tipo *Collector.*
- Una istanza del *Collector* viene ridotta alla *LinkedList*<Persone>

```
Collector<Product, ?, LinkedList<Product>> toLinkedList =
    Collector.of(LinkedList::new, LinkedList::add,
        (first, second) -> {
                first.addAll(second);
                return first;
            });
LinkedList<Product> linkedListOfPersons =
    productList.stream().collect(toLinkedList);
```

Stream Lazy Invocation 1

- Le operazioni intermedie sono ***lazy*** e questo significa che **saranno invocate solo se necessario per l'esecuzione dell'operazione terminale**
- Per esempio, chiamiamo il metodo *wasCalled()* che incrementa un contatore interno ogni volta che è richiamato.

```
private long counter;

private void wasCalled() {
    counter++;
}
```

Stream Lazy Invocation 2

- Ora chiamiamo il metodo *wasCalled()* dall'operazione *filter()*

```
List<String> list = Arrays.asList("abc1", "abc2", "abc3");
counter = 0;
Stream<String> stream = list.stream().filter(element -> {
    wasCalled();
    return element.contains("2");
});
```

Stream Lazy Invocation 3

- Poiché abbiamo un source di 3 elementi, possiamo assumere che il metodo *filter()* venga chiamato tre volte e il valore della variabile *counter* sia tre.
- Tuttavia questo codice NON cambierà il valore di *counter* del tutto e rimarrà zero e il metodo *filter()* non verrà mai chiamato
- Questo avviene perché non è presente l'operazione terminale
- Si può riscrivere il codice con una operazione *map()* e una operazione terminale *findFirst()*.
- Si può anche inserire un log per verificare la sequenza delle chiamate

Stream Lazy Invocation 4

```
Optional<String> stream = list.stream().filter(element -> {
    log.info("filter() was called");
    return element.contains("2");
}).map(element -> {
    log.info("map() was called");
    return element.toUpperCase();
}).findFirst();
```

- Il metodo *filter()* viene chiamato due volte e il metodo *map()* una sola volta
- Questo in quanto la «pipeline» esegue verticalmente e nell'esempio il primo elemento dello stream non soddisfa il predicato del filtro
- L'operazione *findFirst()* è valida per giusto un elemento e la Lazy invocation permette di evitare due chiamate al metodo, una per il *filter()* e una per la *map()*.

Stream Execution Order

- Da fare

Stream Parallel

- Da fare

Stream Generation

Con Java 8, l'interface Collections ha due metodi per generare uno Stream

- **Stream()**
  Restituisce uno Stream sequenziale a partire dalla Collection come source
- **parallelStream()**
  Restituisce un Parallel Stream sequenziale a partire dalla Collection come source

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream()
                    .filter(string -> !string.isEmpty()).collect(Collectors.toList());
```

Java Generating forEach

- Stream fornisce il metodo '**forEach**' per iterare ciascun elemento dello Stream.
- Il codice seguente mostra come stampare 10 numeri random usando forEach

```
Random random = new Random();
random.ints().limit(10).forEach(System.out::println);
```

Java Generating Stream map

- Il metodo '**map**' è usato per **mappare** ciascun elemento al suo **risultato** corrispondente
- Il codice seguente mostra come quadrati di numeri usando map

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5); //get list of unique

 squares List<Integer> squaresList = numbers.stream()
                                .map( i -> i*i)
                                .distinct()
                                .collect(Collectors.toList());
```

Java Generating Stream filter

- Il metodo '**filter**' è usato per **eliminare** elementi basati su un criterio
- Il codice seguente produce un counter di stringhe vuote usando filter

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
int count = strings.stream().filter(string -> string.isEmpty()).count();
```

Java Generating Stream limit

- Il metodo '**limit**' è usato per **ridurre** le dimensioni dello Stream
- Il codice seguente mostra come stampare 10 numeri random usando limit

```
Random random = new Random();

random.ints().limit(10).forEach(System.out::println);
```

Java Generating Stream sorted

- Il metodo '**sorted**' è usato per **ordinare** gli elementi dello stream
- Il codice seguente mostra come stampare 10 numeri random usando limit e con un risultato sortato

```
Random random = new Random();

random.ints().limit(10).sorted().forEach(System.out::println);
```

Java Generating Stream Parallel Processing

- Il metodo '**parallelStram**' è l'alternativa di stream per elaborazioni parallele
- Il codice seguente mostra come stampare un count di stringhe vuote usando *parallelStream*
- Operazione semplice passare da stream sequenziali a paralleli

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");

//get count of empty string
long count = strings.parallelStream()
                    .filter(string -> string.isEmpty())
                    .count();
```

Java Generating Stream Collectors

- I '**collectors**' sono usati per combinare il risultato di una elaborazione sugli elementi di uno Stream, per produrre il risultato finale
- I Collectors possono essere usati per restituire una lista, una map,una stringa, un count, una sum, etc.

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg", "abcd","", "jkl");
List<String> filtered = strings.stream()
                               .filter(string -> !string.isEmpty())
                               .collect(Collectors.toList());
System.out.println("Filtered List: " + filtered);
String mergedString = strings.stream()
                               .filter(string -> !string.isEmpty())
                               .collect(Collectors.joining(", "));
System.out.println("Merged String: " + mergedString);
```

Java Generating Stream Statistics

- Con Java 8, i collectors statistics sono stati introdotti per calcolare tutte le statistiche una volta terminata l'elaborazione dello stream

```
List numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
IntSummaryStatistics stats = numbers.stream()
                                    .mapToInt((x) -> x)
                                    .summaryStatistics();
System.out.println("Highest number in List : " + stats.getMax());
System.out.println("Lowest number in List : " + stats.getMin());
System.out.println("Sum of all numbers : " + stats.getSum());
System.out.println("Average of all numbers : " + stats.getAverage());
```

Java Stream Interface Methods

Methods	Description
boolean allMatch (**Predicate**<? **super T**> *predicate*)	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
boolean anyMatch (**Predicate**<? **super T**> *predicate*)	It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.
static <T> **Stream.Builder**<T> **builder**()	It returns a builder for a Stream.

Java Stream Interface Methods

Methods	Description
<R,A> R collect(Collector<? super T,A,R> collector)	It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.

Java Stream Interface Methods

Methods	Description
boolean allMatch(Predicate<? super T> predicate)	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
boolean anyMatch(Predicate<? super T> predicate)	It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.
static <T> Stream.Builder<T> builder()	It returns a builder for a Stream.

Java Stream Interface Methods

Methods	Description
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)	It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.

Java Stream Interface Methods

Methods	Description
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)	It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.

Java Stream Interface Methods

Methods	Description
long count()	It returns the count of elements in this stream. This is a special case of a reduction.
Stream<T> distinct()	It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
static <T> Stream<T> empty()	It returns an empty sequential Stream.

Java Stream Interface Methods

Methods	Description
Stream<T> filter(Predicate<? super T> predicate)	It returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T> findAny()	It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T> findFirst()	It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned.

Java Stream Interface Methods

Methods	Description
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)	It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

Java Stream Interface Methods

Methods	Description
DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)	It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.)

Java Stream Interface Methods

Methods	Description
IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)	It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

Java Stream Interface Methods

Methods	Description
LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)	It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)

Java Stream Interface Methods

Methods	Description
void forEach(Consumer<? super T> action)	It performs an action for each element of this stream.
void forEachOrdered(Consumer<? super T> action)	It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
static <T> Stream<T> generate(Supplier<T> s)	It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.

Java Stream Interface Methods

Methods	Description
static <T> Stream<T> iterate(T seed,UnaryOperator<T> f)	It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
Stream<T> limit(long maxSize)	It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<R> Stream<R> map(Function<? super T,? extends R> mapper)	It returns a stream consisting of the results of applying the given function to the elements of this stream.

Java Stream Interface Methods

Methods	Description
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)	It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream mapToInt(ToIntFunction<? super T> mapper)	It returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream mapToLong(ToLongFunction<? super T> mapper)	It returns a LongStream consisting of the results of applying the given function to the elements of this stream.

Java Stream Interface Methods

Methods	Description
Optional<T> max(Comparator<? super T> comparator)	It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction.
Optional<T> min(Comparator<? super T> comparator)	It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction.
boolean noneMatch(Predicate<? super T> predicate)	It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.

Java Stream Interface Methods

Methods	Description
@SafeVarargs static <T> Stream<T> of(T... values)	It returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T> of(T t)	It returns a sequential Stream containing a single element.
Stream<T> peek(Consumer<? super T> action)	It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

Java Stream Interface Methods

Methods	Description
Optional<T> reduce(BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T reduce(T identity, BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.

Java Stream Interface Methods

Methods	Description
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T> skip(long n)	It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned.

Java Stream Interface Methods

Methods	Description
Stream<T> sorted()	It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
Stream<T> sorted(Comparator<? super T> comparator)	It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[] toArray()	It returns an array containing the elements of this stream.

Java Stream Interface Methods

Methods	Description
<A> A[] toArray(IntFunction<A[]> generator)	It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.