Streams

In Java 8

Java Features

Presented by Ramin Zare

بخش اول

1. Stream ها چه هستند؟
2. تفاوت بین collection ها و Stream ها

Stream ها چه هستند ؟

- یک Stream ترتیبی از عناصر اطلاعاتی است که یک سری عملیات aggregation را میتواند به صورت sequential یا parallel انجام دهد
- یک عملیات aggregation به عملکردی گفته میشود که یک مقدار واحدی را از یک مجموعه اطلاعات محاسبه میکند
- تفاوت یک Stream با یک Collection پس در چیست ؟
  - هر دوی آنها abstraction از یک مجموعه داده هستند
  - Collection ها روی ذخیره سازی و نحوه ی آن تمرکز دارند
  - Stream ها روی نحوه محاسبات روی عناصر یک دیتا سورس تمرکز دارند

تفاوت Stream ها با Collection ها

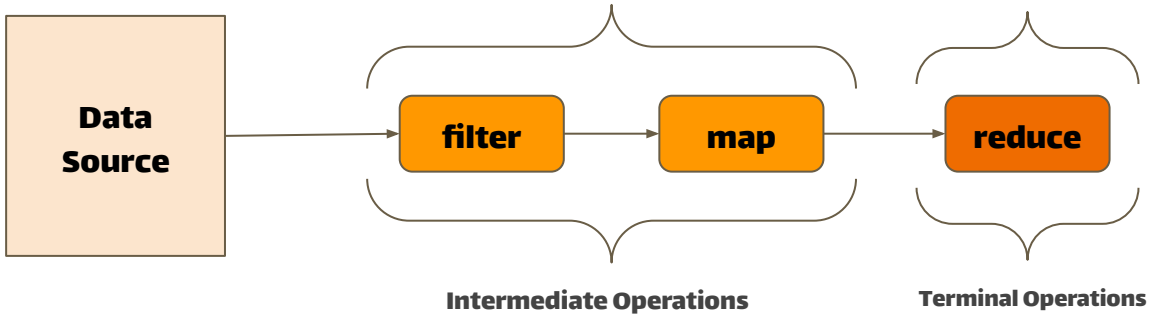
- Stream ها هیچ فضای ذخیره سازی ندارند
- Stream ها میتوانند نمایشگر یک ترتیبی از عناصر نامتناهی باشند
- طراحی Stream ها بر مبنای iteration داخلی است
- آنها طوری طراحی شده اند که بتوانند بدون نیاز به کد اضافه ای توسط برنامه نویس parallel کار کنند
- طراحی آنها به طوری است که از functional programming پشتیبانی کنند
- Stream ها از عملیات Lazy پشتیبانی میکنند
- Stream ها میتوانند مرتب شده یا نامرتب باشند
- آنها قابلیت استفاده مجدد ندارند

بخش دوم

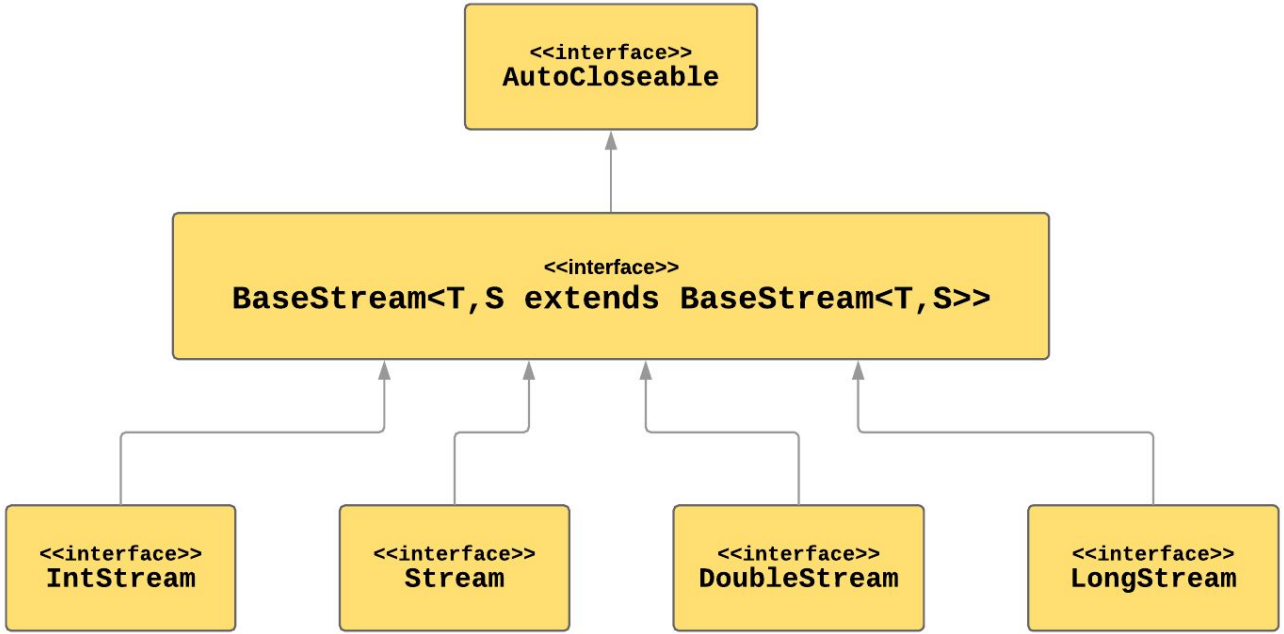
1. Stream Operation
2. API
   1. کلاس BaseStream API
3. ساخت Stream ها
   1. از روی مقادیر
   2. تهی

Stream Operation

- استریم ها دو نوع عملیات را پشتیبانی میکنند:
  - Intermediate Operations
    - این نوع عملیات را Lazy نیز می نامند
  - Terminal Operations
    - این نوع عملیات را eager نیز می نامند



API


<<interface>>
AutoCloseable
<<interface>>
BaseStream<T,S extends BaseStream<T,S>>
<<interface>>
IntStream
<<interface>>
Stream
<<interface>>
DoubleStream
<<interface>>
LongStream

BaseStream کلاس API

- `Iterator<T> iterator()`
- `S sequential()`
- `S parallel()`
- `boolean parallel()`
- `S unordered()`
- `void close()`
- `S onClose(Runnable closeHandler)`

ساخت Stream ها

- از روی مقادیر
- Empty Streams (تهی)
- توسط فانکشن ها
- از روی آرایه ها
- از روی Collection ها
- از روی فایل ها
- از روی منابع دیگر ...

ساخت Stream ها
از روی مقادیر

- `<T> Stream<T> of(T t)`
- `<T> Stream<T> of(T... values)`
- `<T> Stream<T> ofNullable(T t)`

ساخت Stream ها
از روی مقادیر

- `Stream.Builder<T>`
  - `void accept(T t)`
  - `Stream.Builder<T> add(T t)`
  - `Stream<T> build()`

IntStream

- `IntStream of(int value)`
- `IntStream of(int... values)`
- `IntStream range(int start,int end)`
- `IntStream rangeClosed(int start, int end)`

Empty Streams

- `Stream<String> stream = Stream.empty();`
- `IntStream numbers = IntStream.empty();`

بخش سوم

1. ساخت Stream ها
    1.1. توسط فانکشن ها
    1.2. از روی آرایه ها
    1.3. از روی Collection ها
    1.4. از روی فایل ها
    1.5. از روی منابع دیگر ...

Streams های بی نهایت از روی فانکشن

یک Stream بی نهایت یک Stream است که data source آن قابلیت ایجاد یک مجموعه بی نهایتی از اعداد را به شما میدهد .

- `<T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)`
- `<T> Stream<T> iterate(T seed, UnaryOperator<T> f)`
- `<T> Stream<T> generate(Supplier<? extends T> s)`

Streams های بی نهایت از روی فانکشن

```
<T> Stream<T> iterate(T seed,
                      Predicate<? super T> hasNext,
                      UnaryOperator<T> next)
```

```
for(    int index= seed;
        hasNext.text(index);
        index = next.ApplyAsInt(index)
    )
```

Streams های بی نهایت از روی فانکشن

- قابلیت ایجاد محدودیت روی Stream ها

```
Stream.iterate(1L , n-> n+2)
        .skip(200)
        .limit(5)
        .forEach(System.out::println);
```

Stream ها از روی آرایه ها

```
IntStream numbers = Arrays.stream(new int[]{1 ,2 ,3})

Stream<String> strings =
    Arrays.stream(new String[]{“Str1” , “Str2”})
```

Stream ها از روی Collection ها

```
List.of(“Str1”,”Str2”).stream();
List.of(“Str1”,”Str2”).parallelStream();
```

Stream ها از روی فایل ها

- در ورژن جاوا ۸ در پکیج هایی مثل java.io , java.nio.file امکاناتی برای استفاده از Stream ها در حین خواندن فایل ها اضافه شده است.
- Stream هایی برای :
  - خواندن خط به خط اطلاعات فایل
  - JarEntry ها از یک JarFile
  - محتویات داخل یک directory
  - جریانی از Path که حاصل جستجوی روی یک directory خاص میباشد
  - جریانی از Path حاصل از ساختار درختی یک directory میباشد

Stream ها از روی دیگر منابع

- این دو متود نیز در جاوا ۸ اضافه شده اند:
  - `chars()`
    - در کلاس CharSequence این متود تمامی character های یک رشته را در قالب Stream برمیگرداند
  - `slipAsStream(String str)`
    - در کلاس java.util.regex.Pattern این متود Stream از حروف مطابق با الگو برمیگرداند

بخش چهارم

1. Optional Utility
2. Optional Methods
3. New Features in Java 9
4. Other Optional Classes

Optional Utility

- در جاوا null به معنای مقدار "هیچ" تلقی میشود . همین موضوع باعث به وجود آمدن خطاهای مکرر NullPointerException میباشد.
- java.util.Optional برای رفع این مسئله معرفی شده است. Factory متدهای این کلاس :
  - <T> Optional<T> empty()
  - <T> Optional<T> of(T value)
  - <T> Optional<T> ofNullable(T value)

Optional Methods

- متد های این کلاس عبارتند از :
  - boolean **isPresent**()
  - void **ifPresent**(Consumer<? super T> action)
  - T get()
  - T **orElse**(T defaultValue)
  - T **orElseGet**(Supplier<? extends T> defaultSupplier)
  - <X extends Throwable> T orElseThrow(Supplier<? Extends X> exceptionSupplier)

Optional Methods in Java 9

- **متد های این کلاس عبارتند از :**
  - void **ifPresentOrElse**(Consumer<? super T> action, Runnable emptyAction)
  - Optional<T> **or**(Supplier<? extends Optional<? extends T>> supplier>
  - Stream<T> **stream**()

Other Optionals

- java.util.OptionalInt
- java.util.OptionalLong
- java.util.OptionalDouble

بخش پنجم

1. Intermediate Operations
2. Terminal Operations
3. Debugging a Stream
4. Filtering
5. Foreach
6. Map & Flat Maps

Intermediate Operations

`distinct`	`peek`
`filter`	`skip`
`flatMap`	`dropWhile (+9)`
`limit`	`takeWhile (+9)`
`map`	`sorted`

Terminal Operations

boolean allMatch	**void** forEach
boolean anyMatch	**T** reduce
Optional findAny	count - min - max
Optional findFirst	**T** collect
boolean noneMatch	

Debugging a Stream with peek

- متد peek یک intermediate operation میباشد که در خلال یک pipeline آخرین تغییرات را در اختیار شما میگذارد ؛ با آن میتوانید عملیات debugging انجام دهید

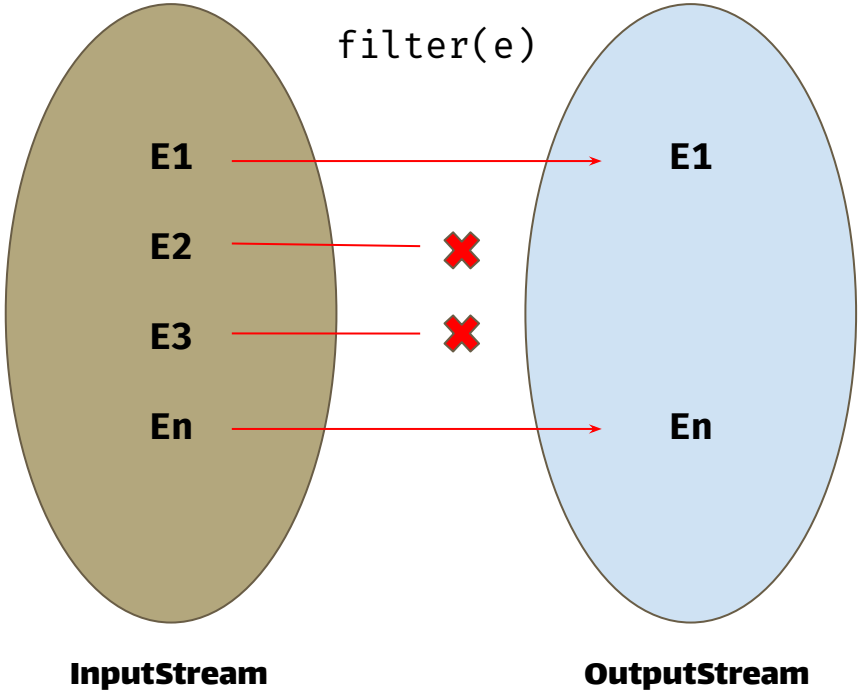
```
peek(Consumer<? super T> action)
```

forEach Operation

- متد forEach یک Terminal operation میباشد که به ازای هر element در پایان یک Stream میتواند یک عملیاتی را انجام دهد

```
void forEach(Consumer<? super T> action)

void forEachOrdered(Consumer<? super T> action)
```

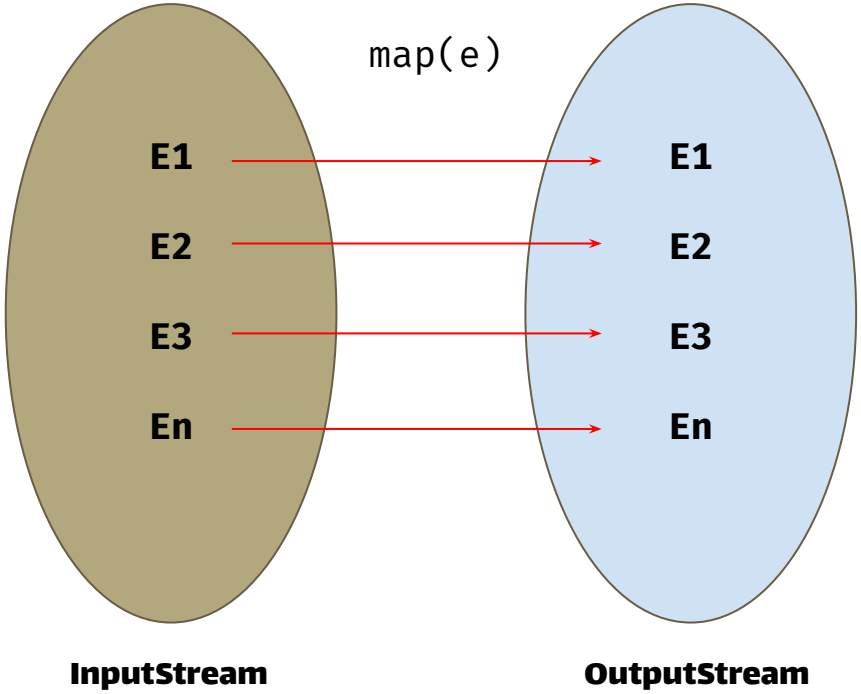
filter operation



Operations to help filtering

```
Stream<T> skip(long count)

Stream<T> limit(long maxCount)

default Stream<T> dropWhile(Predicate<? super T> predicate)

default Stream<T> takeWhile(Predicate<? super T> predicate)
```

map Operation



map Operation

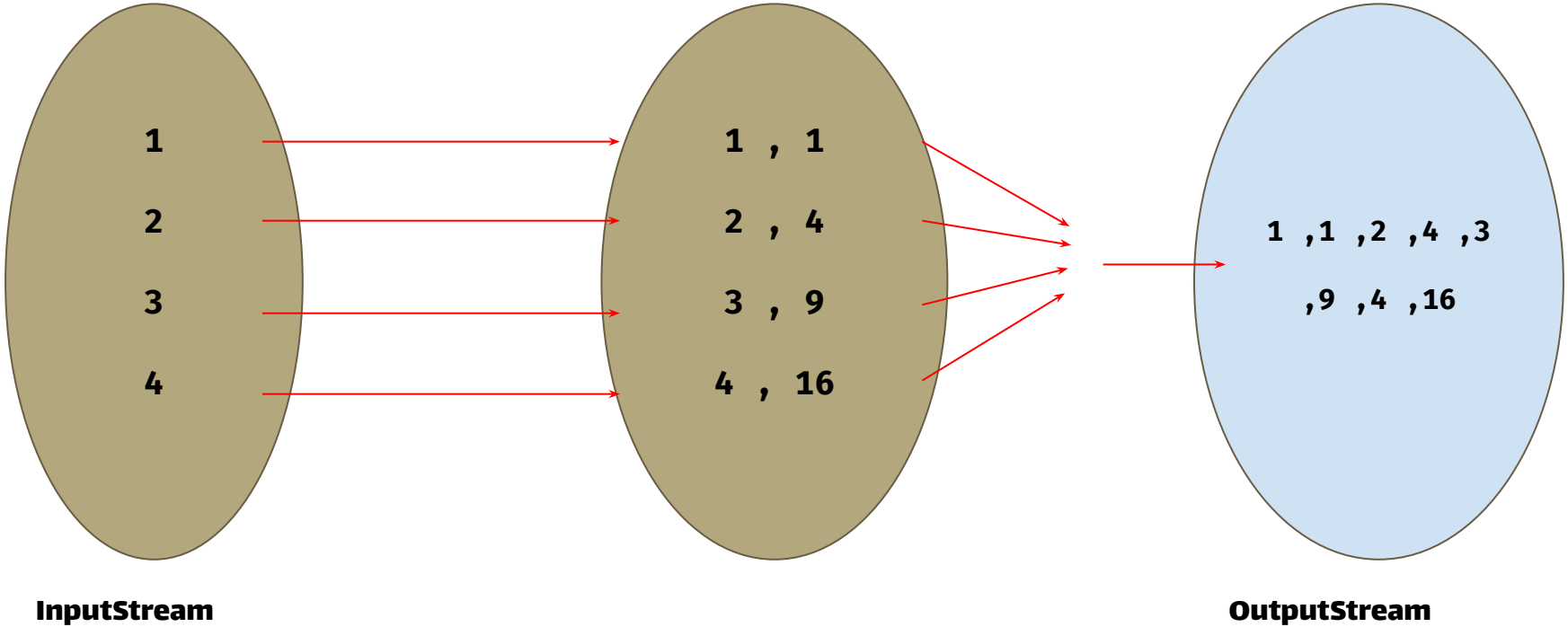
```
<R> Stream<R> map(Function<? super T,? extends R> mapper)

DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)

IntStream mapToInt(ToIntFunction<? super T> mapper)

LongStream mapToLong(ToLongFunction<? super T> mapper)
```

IntStream :

```
<U> Stream<U> mapToObj(IntFunction<? extends U> mapper)
```

Flattening Stream (One To Many Mapping)

```
flatMap(e -> Stream.of(e , e ^ 2))
```



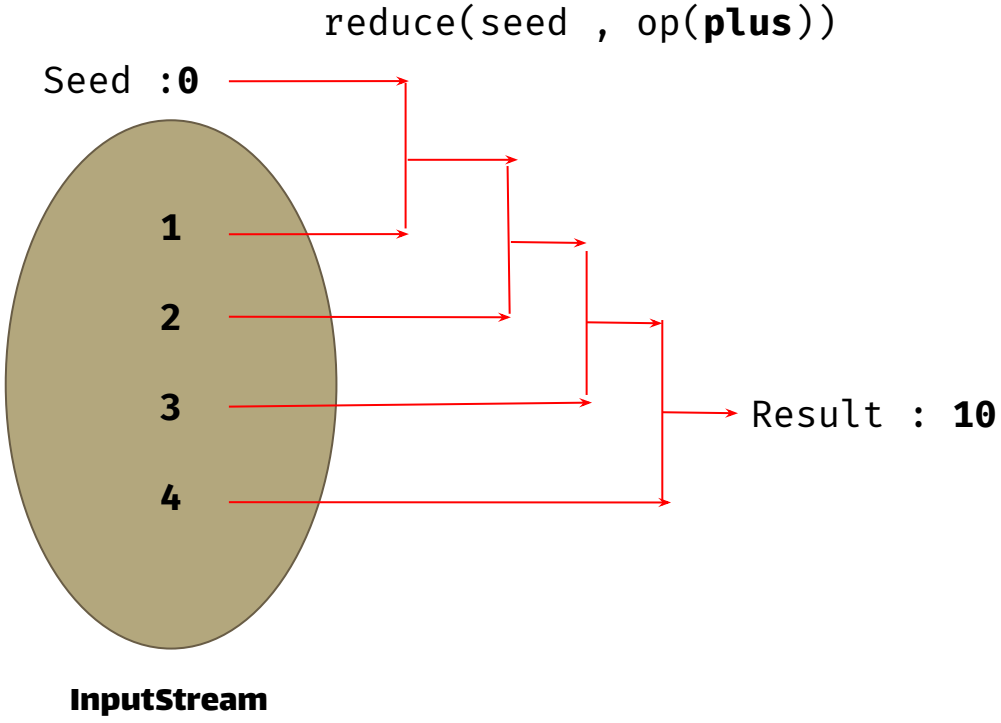
flatMap Example

```
long count = Stream.of("String1","String2")
        .map(name -> name.chars())
        .flatMap(intStream -> intStream.mapToObj(n->(char)n))
        .filter(ch -> ch == 'i')
        .count();
```

بخش ششم

1. **terminal operator**
    1.1. reduce
    1.2. sum - max - average - min - count
    1.3. Collect
        1.3.1. Collectors
        1.3.2. Collecting Map
        1.3.3. Joining String
        1.3.4. Summary Statistics

reduce terminal operation



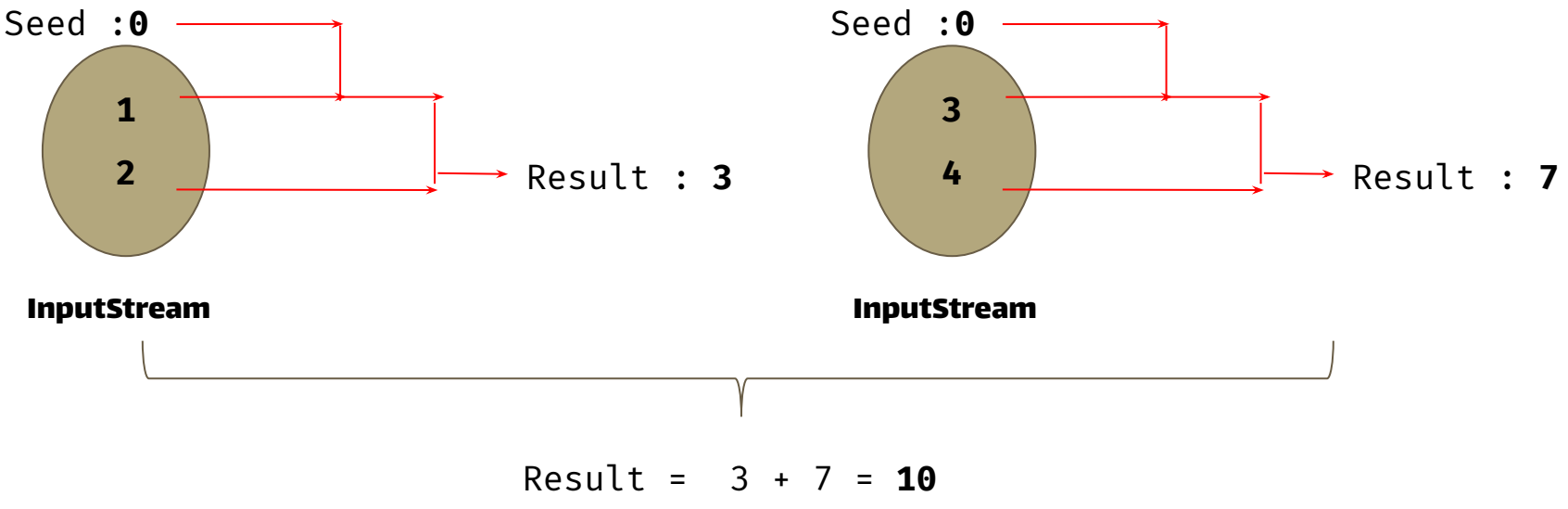
reduce terminal operation

- `T reduce (T identity , BinaryOperator<T> accumulator)`
- `Optional<T> reduce(BinaryOperator<T> accumulator)`
-

```
<U> U reduce(U identity ,
        BiFunction<U,? super T,U>  accumulator,
        BinaryOperator<U> combiner)
```

Parallel reduce terminal operation

```
reduce(seed , op(plus) , combiner(plus))
```



Other reduction operators

- `sum`
- `max`
- `average`
- `min`
- `count`

collect terminal operation

- در حالت های مختلفی نیاز داریم که خروجی stream را در قالب collection هایی مثل List, Set یا Map دریافت کنیم. ممکن است که حتی بخواهید منطق پیچیده ای برای خروجی گرفتن پیاده سازی کنید. مثلا group کردن آنها
- تمامی اینها را با operation collect انجام می پذیرد

- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator,BiConsumer<R,R> combiner>`
- `<R,A> R collect(Collector<? super T,A,R> collect)`

Collector interface

```
public interface Collector<T, A, R> {

   Supplier<A> supplier();

   BiConsumer<A, T> accumulator();

   BinaryOperator<A> combiner();

   Function<A, R> finisher();

   Set<Characteristics> characteristics();

}
```

Collectors helper class

- به دلیل اینکه ساخت Collector ها زمانبر و کمی پیچیده است در جاوا ۸ یک مجموعه آماده ای از آنها از قبل تدارک دیده شده است:
  - Collectors.toList()
  - Collectors.toSet()
  - Collectors.toCollection(TreeSet::new)
  - Collectors.counting()

Collecting Map

```
Collectors.toMap(
    Function<? super T,? extends K> keyMapper(1),
    Function<? super T,? extends U> valueMapper(1),
    BinaryOperator<U> mergeFunction(2),
    Supplier <M> mapSupplier(3)
)
```

Joining Strings

```
Collectors.joining()

Collectors.joining(CharSequence delimiter)

Collectors.joining(CharSequence delimiter,
                   CharSequence prefix,
                   CharSequence suffix)
```

Summary Statistics

- ممکن است در مواردی نیاز داشته باشید که از یک stream اطلاعات max , min , avg, sum و count را یکجا داشته باشید. یکی از کلاس های زیر می تواند در نگهداری این اطلاعات به شما کمک کند:
  - DoubleSummaryStatistics
  - LongSummaryStatistics
  - IntSummaryStatistics
- همچنین با استفاده از Collectors.summarizingDouble میتوان چنین خروجی را از Stream ها دریافت کرد

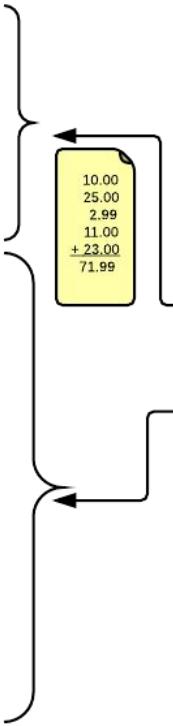
بخش هفتم - بخش آخر

1. **terminal operator**
    1.1. Collect
        1.1.1. Grouping Data
        1.1.2. Partitioning Data
    1.2. Finding and matching

Grouping Data

transactions

id	paid_out	description
1	10.00	Amazon
2	25.00	Amazon
3	2.99	Amazon
4	11.00	Amazon
5	23.00	Amazon
6	25.00	Michaels
7	300.00	Michaels
8	1.00	Michaels
9	11.00	Michaels
10	2.99	Michaels
11	25.00	Michaels
12	1.00	Michaels
13	25.00	Michaels
14	10.00	Michaels




```
SELECT
  SUM(paid_out) AS paid_out_total,
  description
FROM transactions
GROUP BY description;
```

paid_out_total	description
71.99	Amazon
400.99	Michaels

Grouping data

```
Collectors.groupBy(
    Function<? super T,? extends K> classifier(1),
    Supplier <M> mapSupplier(3)
    Collector<? super T,A,D> downstream(2))
```

Partitioning Data

```
Collectors.partitioningBy(
    Predicate<? super T,? extends K> predicate(1),
    Collector<? super T,A,D> downstream(2))
```

Finding and matching

```
boolean allMatch(Predicate<? super T> predicate)

boolean anyMatch(Predicate<? super T> predicate)

boolean noneMatch(Predicate<? super T> predicate)

Optional<T> findAny()

Optional<T> findFirst()
```

Thanks!

Presented by :

Ramin Zare