Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
**Istituto sistemi informativi e networking**



**SUPSI**

# Streams

Object Oriented Programming

Tiziano Leidi

12.11.2021

# Streams

Introduced in Java 8, the Stream API is used to process collections of objects.

Streams should not be confused with Java I/O streams: the *InputStream* and *OutputStream* classes used in Java for I/O.

Streams relate instead to collections: *List*, *Set*, *Map* (and also to arrays or generator functions).

# Streams

Streams are sequences of elements, but unlike collections, streams do not contain the elements directly.

A stream does not store data. It also never modifies the underlying data source.

Streams transport the elements from a source through sequences of functions that transform them.

# Streams

Practically speaking, the functionality provided in Java for streams allows writing operations on collections (and other data sources) at a high level of abstraction.

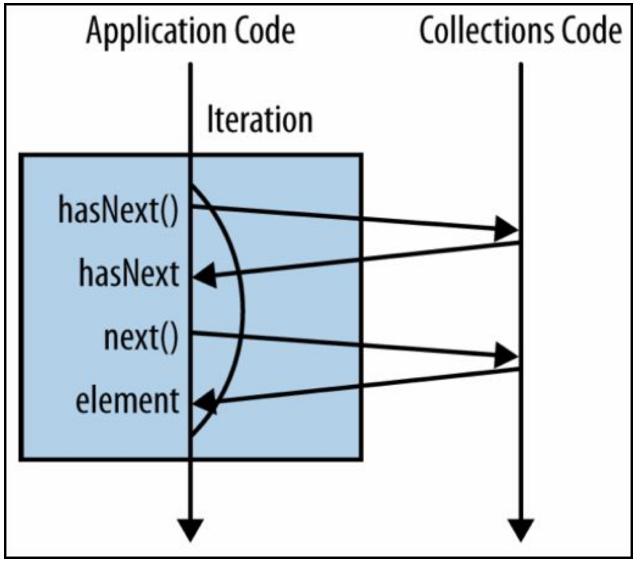
Simply put, streams are wrappers around a data source, allowing to operate with that data source and making bulk processing convenient and fast.

# Implicit Iteration

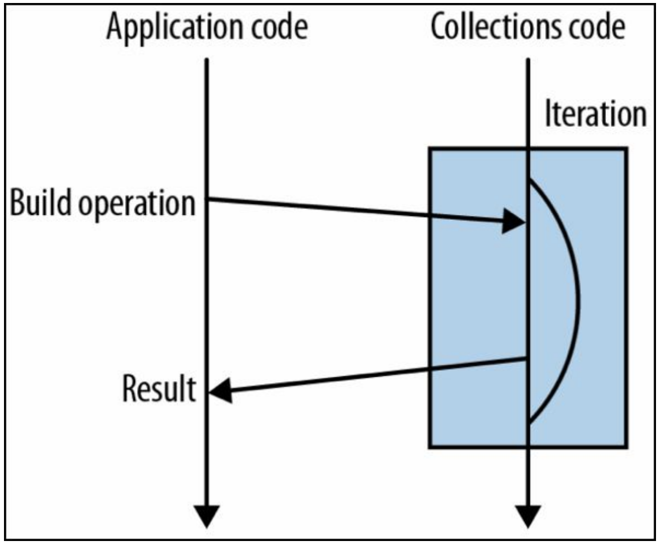
In particular, with streams it is possible to replace explicit iteration with implicit iteration (also called internal iteration).

```java
long count = 0;
for (Planet planet : planets)
   if (planet.isGasGiant())
       count++;
```

```java
long count = 0;
Iterator<Planet> iterator = planets.iterator();
while (iterator.hasNext()) {
   Planet planet = iterator.next();
   if (planet.isGasGiant())
       count++;
}
```



```java
long count = planets.stream().filter(planet -> planet.isGasGiant()).count();
```



# Implicit Iteration

So, a stream is the internal equivalent of an iterator.

With internal iterations, the control of the iteration is delegated to the stream (thus, at practical level, to the source collection).

# Stream Creation

Streams can be obtained:

- from collections with the *.stream()* method
- by converting an array with *Stream.of()* or *Arrays.stream()*
- from individual objects, using *Stream.of()*
- using a *Stream.Builder*

```java
String[] values = {"one", "two", "three"};
Stream<String> stream1 = Stream.of(values);


Stream<String> stream2 = Stream.of("one", "two", "three");


Stream.Builder<String> builder = Stream.builder();

builder.accept("one");
builder.accept("two");
builder.accept("three");

Stream<String> stream3 = builder.build();
```

# Infinite Streams

Sometimes, it might be needed to perform operations while the elements are still getting generated.

Infinite streams (also called unbounded streams) can be generated with:

- *Stream.generate()*: a *Supplier* needs to be provided which gets called whenever new stream elements need to be generated
- *Stream.iterate()*: takes an initial value, called seed and a function that generates the next value using the previous one

```java
Stream<Double> rndNumStream = Stream.generate(Math::random);

Stream<Integer> evenNumStream = Stream.iterate(2, i -> i * 2);
```

# Higher Order Functions

All the functions made available by streams are called higher-order functions (or alternatively aggregate operations).

An higher-order function is a function that accepts another function as a parameter, or returns a function as a value.

Java streams provide: min, max, filter, map, collect(), ...

# Pipelines

Streams allow to chain more than one higher-order function after the other in a pipeline.

A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

Intermediate operations return a new stream on which further processing can be done. Terminal operations mark the stream as consumed, after which point it can no longer be used further.

```java
Set<AtmosphereType> types = solarSystem
                            .getPlanets()
                            .stream()
                            .filter(planet -> planet.hasRings())
                            .map(planet -> planet.getAtmospereType())
                            .collect(Collectors.toSet());
```

# Lazy vs. Eager Evaluation

An important advantage of internal iteration, is that streams have freedom on how to execute the iteration.

For example, streams work most of the time lazily, performing operations only when the result is actually required.

# Lazy vs. Eager Evaluation

```
solarSystem.getPlanets().stream()
                        .filter(planet -> planet.hasRings())
                        .collect(Collectors.toList());
```

the call to *filter()* builds up a stream recipe, but there's nothing to force this recipe to be immediately executed. These type of functions have therefore a lazy behaviour.

Instead, methods such as *collect()* that generate a final value out of the stream sequence, need to execute the recipe and have therefore eager behaviour.

# Lazy vs. Eager Evaluation

In principle, the behaviour is as follows:

- computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.
- all intermediate operations are lazy, so they're not executed until a result of a processing is actually needed.

This behavior becomes important when the input stream is infinite.

# Stream Functions

The following slides provide descriptions and examples of the most common stream functions provided by Java.

# ForEach

ForEach is the simplest and most common operation. It loops over the stream elements, calling the supplied *Consumer* on each element.

```
employees.stream().forEach(curEmployee ->
                          curEmployee.salaryIncrement(10.0));
```

ForEach is a terminal operation.

# Peek

Peek is an intermediate operation, similar to ForEach, that performs the specified operation on each element of the stream and returns a new stream that can be used further.

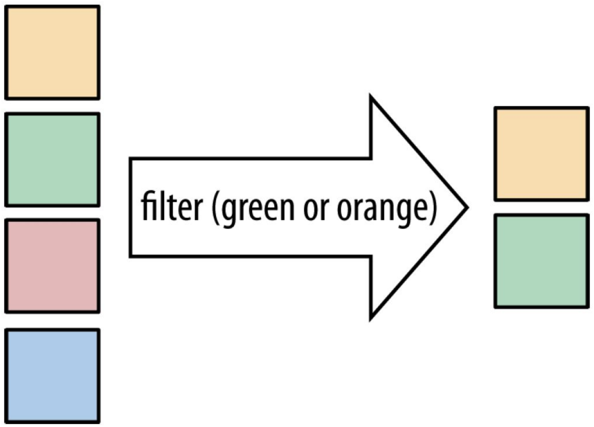
```
employees.stream()
        .peek(curEmployee -> curEmployee.salaryIncrement(10.0))
        .peek(System.out::println)
        .collect(Collectors.toList());
```

The functional interface of Peek is the *Consumer*.

# Filter

When you need to loop over some data and check each element, use the filter() method:

```
Stream<ElementType> filteredElements =
              elements.stream().filter(element -> check(element));
```




The functional interface for filter() is the *Predicate*.

# Filter

Filter is a lazy operation, equivalent to a loop over a collection, with an if statement that retain some elements, while throwing others out:

```
Collection<ElementType> filteredElements = //...
for(ElementType element : elements)
   if(check(element))
       filteredElements.add(element);
```

The presence of an if statement in the middle of a for loop is a pretty strong indicator that it is possible to use filter().

# Collect

Collect is an eager operation that generates a *Collection* from the values in a *Stream*:

```
Stream<ElementType> filteredElements = //...
List<ElementType> listOfElements =
                 filteredElements.collect(Collectors.toList());
```

The provided argument has to be of type *Collector*.

# Collectors

Collectors is a class that provides factory methods for instantiating objects of type Collector.

The most frequently used types of collectors are *Collectors.toList()* and *Collectors.toSet()*, but many others are provided.

# Count, Min and Max

Count, Min and Max are also eager operations with specific functionality:

- *count()* counts how many objects are in a given stream.
- *min()* allows finding the smallest element.
- *max()* allows finding the largest element.

The functional interface for min() and max() is the *Comparator*. Conveniently, the class Comparator provides factory methods for building comparators:

```
elements.stream().min(Comparator.comparing(element ->
                                           element.getProperty()));
```

# Sorted and Distinct

Sorted sorts the stream elements based on the *Comparator* passed into it.

Distinct does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements.

# AllMatch, AnyMatch, and NoneMatch

These operations take a *Predicate* and return a boolean:

- allMatch() checks if the predicate is true for all the elements in the stream.
- anyMatch() checks if the predicate is true for any one element in the stream.
- noneMatch() checks if there are no elements matching the predicate.

Processing is stopped as soon as the answer is determined.

# FindFirst

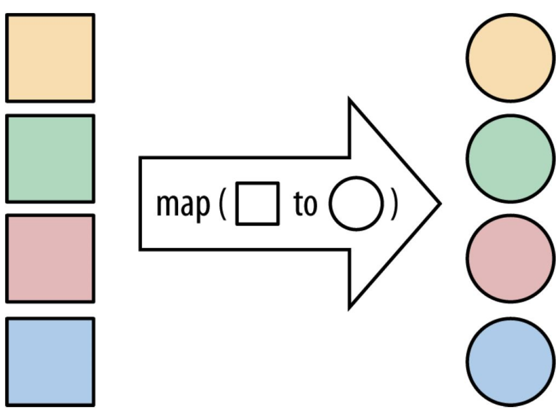
FindFirst is a terminal operation that returns an Optional for the first entry in the stream. the Optional can, of course, be empty.

```
employee = Stream.of(ids)
        .map(employeeRepository::findById)
        .filter(curEmployee -> curEmployee != null)
        .filter(curEmployee -> curEmployee.getSalary() > 100000)
        .findFirst()
        .orElse(null);
```

# Map

If you need to convert a value of one type into another, map lets you apply this function to a stream of values, producing another stream of the new values.

```
Stream<RelatedElementType> relatedElements =
    elements.stream().map(element -> element.getRelatedElement());
```



# Map

Map is a lazy operation, equivalent to a loop over all the values in the collection, that performs any type of transformation on each element. You would then add each of the resulting values into a new collection.

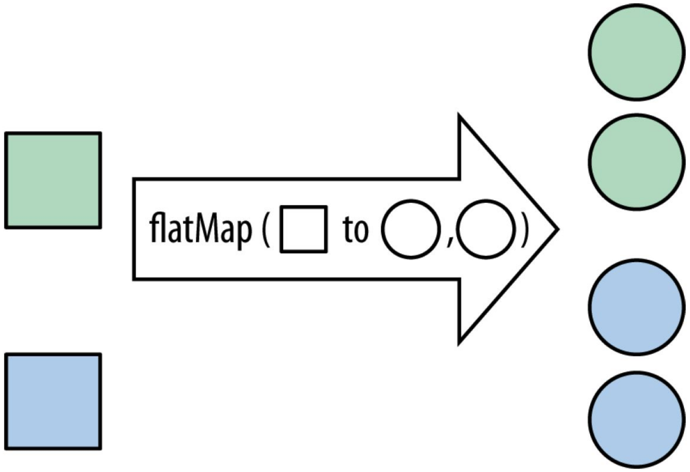
It isn’t necessary for both the argument and the result to be the same type.

The lambda expression passed in must be an instance of the *Function* functional interface.

# FlatMap

FlatMap is similar to Map, but lets you replace each value of the stream with an independent stream, then concatenates all the resulting streams together into a final stream.

If you do the same with map(), you'll end up with a stream of streams. This is where flatMap comes in handy.



# FlatMap

```
Stream<Satellite> allSatellites =
            solarSystem.getPlanets().stream().flatMap(planet ->
            planet.getSatellites());
```

The functional interface required by FlatMap is still a *Function*, but the return type is restricted to streams, instead of any type of value.

# Reduce

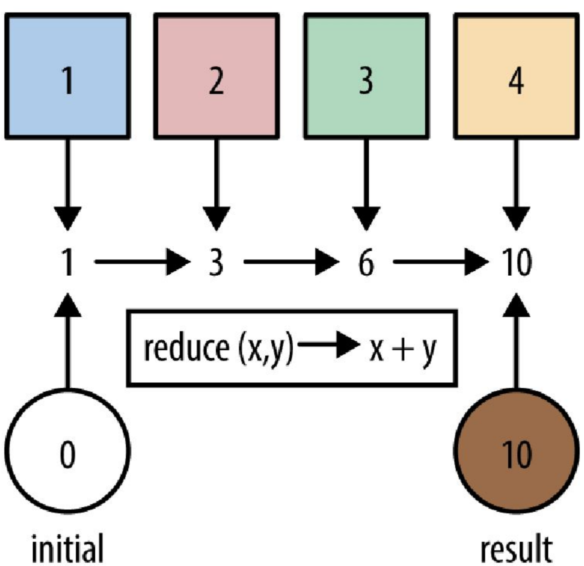
The Reduce operation is of help when you’ve got a stream of values and you want to generate a single result.

Count, Min, and Max are all forms of reduction directly provided by the standard library.

# Reduce

The Reduce operation is eager. The type of the reducer is a *BinaryOperator*.

```
Stream.of(1, 2, 3, 4).reduce(0, (accumulator, element) ->
                                  accumulator + element);
```



# Short-Circuit Operations

Some operations are called short-circuiting operations. Short-circuiting operations allow computations on infinite streams to complete.

```
List<Integer> collect = infiniteStream
        .skip(3)
        .limit(5)
        .collect(Collectors.toList());
```

skip() to skip first 3 elements, and limit() to limit to 5 elements

# Stream Specializations

IntStream, LongStream, and DoubleStream are primitive specializations for *int*, *long* and *double* respectively.

These specialized streams do not extend Stream but extend BaseStream (on top of which Stream is also built). As a consequence, not all operations supported by Stream are present.

# Stream Specializations

The most common way of creating an IntStream, LongStream or DoubleStream is to call *mapToInt(), mapToLong(), mapToDouble()* on an existing stream.

Specialized streams provide additional operations that are convenient when dealing with numbers. For example sum(), average(), range(), ...

```java
Double averageSalary = employees.stream()
        .mapToDouble(Employee::getSalary)
        .average()
        .orElseThrow(NoSuchElementException::new);
```

# Advanced Collect

*summarizingDouble()* is interesting collector, which returns a special class containing statistical information for the resulting values:

```
DoubleSummaryStatistics stats = empList.stream()
        .collect(Collectors.summarizingDouble(Employee::getSalary));

assertEquals(stats.getCount(), 3);
assertEquals(stats.getSum(), 600000.0, 0);
assertEquals(stats.getMin(), 100000.0, 0);
assertEquals(stats.getMax(), 300000.0, 0);
assertEquals(stats.getAverage(), 200000.0, 0);
```

# Advanced Collect

with *partitioningBy()* we can partition a stream into two, based on whether the elements satisfy certain criteria or not:

```
List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);
Map<Boolean, List<Integer>> isEven = intList.stream().collect(
       Collectors.partitioningBy(i -> i % 2 == 0));

assertEquals(isEven.get(true).size(), 4);
assertEquals(isEven.get(false).size(), 1);
```

# Advanced Collect

*groupingBy()* offers advanced partitioning, where we can partition the stream into more than just two groups. It takes a classification function as its parameter. The value returned by the function is used as a key to the resulting map:

```
Map<Character, List<Employee>> groupByAlphabet = empList.stream().collect(
        Collectors.groupingBy(e -> new Character(e.getName().charAt(0))));

assertEquals(groupByAlphabet.get('B').get(0).getName(), "Bill Gates");
assertEquals(groupByAlphabet.get('J').get(0).getName(), "Jeff Bezos");
assertEquals(groupByAlphabet.get('M').get(0).getName(), "Mark Zuckerberg");
```

# Summary

- Introduction to Streams
- Explicit iteration vs. implicit iteration
- Stream creation and infinite streams
- Higher order functions
- Pipelines, intermediary and terminal operations
- Lazy vs. eager evaluation
- Description and examples of common stream functions