Functional Interface

Interface having just one abstract method (default methods may be as many as needed)

Annotation @FinctionalInterface checks requirements of a functional interface

Functional Interface is intended for possible usage of the Lambda expressions and Method References

Lambda Expressions

Instead of a regular class

Instead of an anonymous class

Only inline syntax :

([name,...])->expression|statement|

{statement;...}

list.sort((x,y)->Integer.compare(y,x));

Method Reference

Instead of lambda expression that calls one method with the same parameters order

Static method: <Class name>::<method name>

Non Static method: <Object reference>::<method name>

Constructor: <Class name>::new

Method Reference Examples

list.sort((x,y)->Integer.compare(x,y)) - lambda

list.sort(Integer::compare) - method reference

list.forEach(x->System.out.println(x)) - lambda

list.forEach(System.out::println) - method reference

Functional Streams

Java 8 Stream isn't related to Input/Output

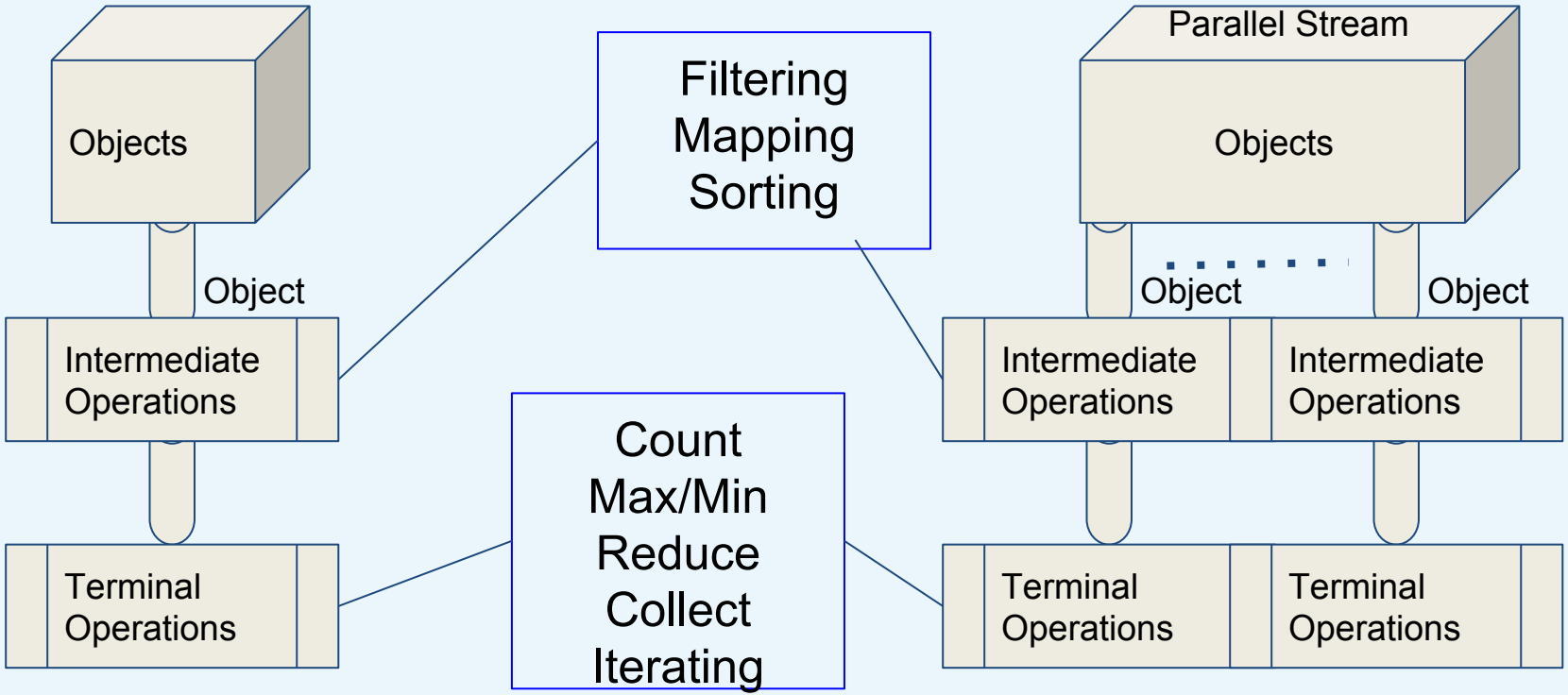
Functional pipeline opening a chain of actions with each object (Monad)

Filter is widespread method name for extracting from collection the objects matching a predicate

Java collection doesn't have the filter method but Stream does. Why?

Parallel streams - parallel performing of a chain actions

Stream Pipeline



How to get Stream Java Core

From collection:

collection.stream()/collection.parallelStream()

From array: *Arrays.stream(array)/Arrays.stream(array).parallel()*

From iterable:

StreamSupport.stream(numbers.spliterator(), false) - regular

StreamSupport.stream(numbers.spliterator(), true) - parallel

From Random Generator:

generator.ints()/generator.longs() (with several options in the method parameters

From BufferedReader:

reader.lines()

Intermediate Operations

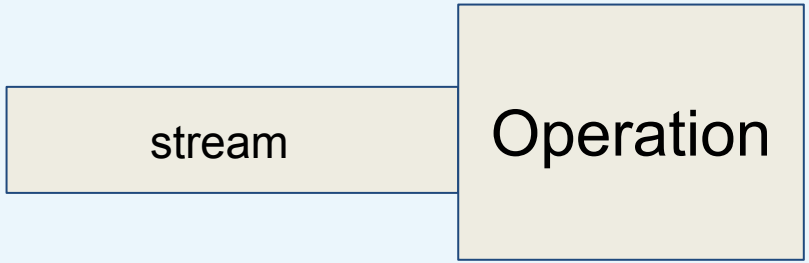
- Laizy - running only after terminal operation
- Return Stream

- ***FIltering***
  - *filter(Predicate<T>)*
- ***Sorting***
  - *sorted()/sorted(Comparator)*

- ***Mapping***
  - Gets a reference to the Function interface
  - *map* - mapping one to one
  - *mapToObj* - mapping one primitive (int,long,double) to an object
  - *mapToInt/Long/Double* - mapping one object to a primitive
  - *flatMap/flatMapToObj/Int/Long/Double* - mapping one to many

Terminal Operations

- Starts running whole pipeline from a getting stream
- Aggregated statistics
- Iterating - forEach
- Reducing
- Collecting



Aggregated Statistics

- For all streams:
  - count/min/max
- For the primitive's streams (IntStream, LongStream, DoubleStream):
  - summaryStatistics() getting reference to Int/Long/DoubleSummaryStatistics class with methods:
    - getMin, getMax, getSum and getAverage

Reducing

- Combines all elements of a stream into a single result
  - For Stream<T> :
    - T reduce (T initial,BiFunction<T,T> accumulator)
    - R reduce(R initial,BiFunction<R,T> accumulator, BiFunction<R,R>combiner)

numbers.stream().reduce(1,(x,y)->x+y)

persons.stream().reduce(0,(sum,p2)->sum+p2.getAge(),(sum1,sum2)->sum1+sum2)

Collecting

- Method ***collect()*** transforms the elements of the stream into a different kind of result, e.g. a List, Set or Map
- Accepts Collector objects
- Pretty complicated but there are the standard collectors for most applied operations:
  - Collectors.toList
  - Collectors.toSet
  - Collectors.groupingBy

GroupingBy

- One of the Collectors (***Collectors.groupingBy()***)
- Groups according to the method ***apply*** of the ***interface Function<T,R>***
- Mostly two kinds:
  - groupingBy(Function<T,R>) returns Map<R,List<T>> - example: lists of the persons (List<T> - List<Person>) having the specific age (R- Integer)
  - groupingBy(Function<T,R>,Collectors.counting()) returns Map<R,Integer> example: how many persons (Integer) have the specific age (R-Integer)