Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Streams Processing Techniques

Source: R. Urma, M. Fusco, A. Mycroft, Java 8 in Action, Manning Publications, 2015

T. Cioara, V. Chifu, C. Pop
2025

Introduction

- Stream - sequence of data elements
  - Supports sequential and parallel operations
- Operation examples
  - Calculate the sum of all elements in a stream of integers
  - Mapping all names in list to their lengths
  - Sort the names in a stream of names
- Java for streams package:
  - java.util.stream (Link)

Interface Summary

Interface
BaseStream<T,S extends BaseStream<T,S>>
Collector<T,A,R>
DoubleStream
DoubleStream.Builder
IntStream
IntStream.Builder
LongStream
LongStream.Builder
Stream<T>
Stream.Builder<T>

Class Summary

Class
Collectors
StreamSupport

Introduction

- **Streams versus Collections**

	Streams	Collections
Elements	Only a part of the Stream is present in memory (its elements are computed when needed)	Holds all elements in memory (the elements may be added and deleted)
Focus	Focus on aggregate computations on data elements from a data source that cold be collection	Focus on storage of data elements for efficient access
Iteration	The iteration is implicit in the operations (streams are smart iterators over collections)	Require explicit iteration over its values

- **Streams are not Collections!**
- **Streams are consuming data from collections, arrays or I/O resources.**

Introduction

- Streams allow writing code that is
  - **Declarative** (concise and reliable)
  - **Composable** (increase flexibility)
  - **Parallelizable** (increase performance)
  - **Pipelined**
    - Many stream operations return a stream thus allowing operations to be **chained** into large pipelines
    - Pipeline enables optimizations such as laziness and short-circuiting
      - can be viewed as database-like query on the data source

Main Features of Streams

- **Declarative Programming**
  - Stream oriented programming using lambda expressions
  - Declarative programming different than the imperative approach
  - Using the declarative style,
    - one says what needs to be done “Find names of three high-calorie dishes.”
    - You don’t implement the filtering (filter), extracting (map), or truncating (limit) functionalities; They’re available through the Streams library
  - Streams API has flexibility to decide how to optimize this pipeline.
    - For example, the filtering, extracting, and truncating steps could be merged into a single pass and stop as soon as three dishes are found

Main Features of Streams

- **Example - immutable class Dish**

```
public class Dish {
 private final String name;
 private final boolean vegetarian;
 private final int calories;
 private final Type type;
 public Dish(String name, boolean
   vegetarian, int calories, Type type) {
     this.name = name;
     this.vegetarian = vegetarian;
     this.calories = calories;
     this.type = type;
 }
 public String getName() { return name;}
 public boolean isVegetarian() {
     return vegetarian;}
 public int getCalories() { return calories;}
 public Type getType() { return type;}
 @Override
 public String toString() { return name; }
 public enum Type { MEAT, FISH, OTHER }
}
```

```
// menu – a list of dishes
List<Dish> menu = Arrays.asList (
   new Dish("pork", false, 800, Dish.Type.MEAT),
   new Dish("beef", false, 700, Dish.Type.MEAT),
   new Dish("chicken", false, 400, Dish.Type.MEAT),
   new Dish("french fries", true, 530, Dish.Type.OTHER),
   new Dish("rice", true, 350, Dish.Type.OTHER),
   new Dish("season fruit", true, 120, Dish.Type.OTHER),
   new Dish("pizza", true, 550, Dish.Type.OTHER),
   new Dish("prawns", false, 300, Dish.Type.FISH),
   new Dish("salmon", false, 450, Dish.Type.FISH));
```

Main Features of Streams

- **Example - Java 7 vs. Java 8**

Java 7

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for (Dish d : menu) {
   if(d.getCalories() < 400)
      // Filter the elements using an accumulator
      lowCaloricDishes.add(d);
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
   public int compare(Dish d1, Dish d2) {
     // Sort the dishes with an anonymous class
     return Integer.compare(d1.getCalories(), d2.getCalories());
   }
});
List<String> lowCaloricDishesName = new  ArrayList<>();
   for(Dish d: lowCaloricDishes)
   lowCaloricDishesName.add(d.getName());
   // Process sorted list to select name of dishes
}
```

Java 8

```
import static java.util.Comparator.comparing;
import static util.stream.Collectors.toList;
    …
List<String> lowCaloricDishesName = menu.stream()
        .filter(d -> d.getCalories() < 400)
        .sorted(comparing(Dish::getCalories))
        .map(Dish::getName)
        .collect(toList());
```

Main Features of Streams

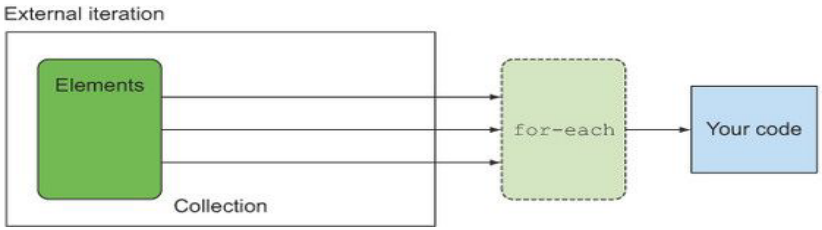
- **Example - discussion**
  - Method **sorted** defined by the interface Stream
    - Stream<T> **sorted** (Comparator<? super T> comparator) - returns a stream consisting of the elements of this stream, sorted according to the provided Comparator
  - Comparator<T> interface defined in Java 8
    - Functional Interface defining
      - abstract **method int compare(T o1, T o2)** and
      - static method (overloaded) **comparing** that takes a Function argument and returns a **Comparator** from object extracted from Function that can be used for sorting purposes
      - … other default and static methods

Main Features of Streams

- **Features**
  - A stream has no storage; it does not store elements
  - A stream pulls (on-demand) its elements from a data source
  - Streams can represent a sequence of infinite elements
  - The design of streams is based on internal iteration
  - Streams are designed to support functional programming
  - Streams are designed to be processed in parallel with no additional work from the developers
  - Streams support lazy operations
  - Streams cannot be reused

Main Features of Streams

- **External (collection) Iteration versus Internal (stream) Iteration**
  - External Iteration
    - Collection specific iteration
      1. Obtain an iterator on a collection,
      2. Process the elements one after the other using the iterator



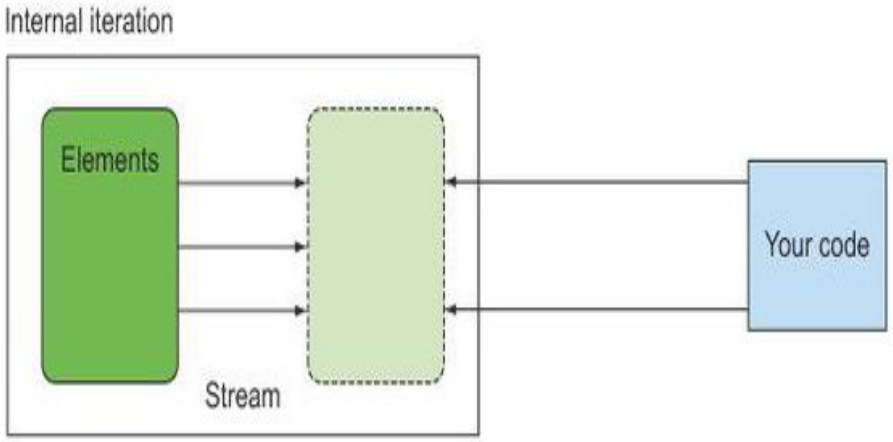
    - Program client pulls values from collection and processes them one by one to get the result
    - Produced a sequential executing code (see the for-each statement) – that can be executed only by one thread

```
// calculate sum of squares of odd numbers
// using an external iterator
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sumSqOdd = 0;
Iterator<Integer> it = numbers.iterator();
while (it.hasNext())  {  //explicit external iterator
   Integer n = it.next();
   if (n % 2 == 1) {
        int square = n * n;
        sumSqOdd = sumSqOdd + square;
   }
}
```

```
// calculate sum of squares of odd numbers
// using an external iterator
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sumSqOdd = 0;
for (int n : numbers) {  // for-each loop iterator
    if (n % 2 == 1) {
        int square = n * n;
        sumSqOdd = sumSqOdd + square;
   }
}
```

Main Features of Streams

- **External (collection) Iteration versus Internal (stream) Iteration**
  - Internal Iteration
    - Uses streams
    - The iteration is achieved internally by the streams



Internal Iteration (stream like)

```
// calculate sum of squares using streams
// (internal iteration)
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sumSqOdd = numbers.stream()
    .filter(n -> n % 2 == 1)
    .map(n -> n * n)
    .reduce(0, Integer::sum);
```

Main Features of Streams

- **Parallel Processing**
  - Modern computers - equipped with multicore processors => parallel processing
  - Java Streams may process the elements in parallel!
  - Streams take care of the details of using the Fork/Join framework internally

```
// uses parallel multithreaded processing
int sumSqOdd = numbers.parallelStream()
                      .filter(n -> n % 2 == 1)
                      .map(n -> n * n)
                      .reduce(0, Integer::sum);
```

Stream Operations

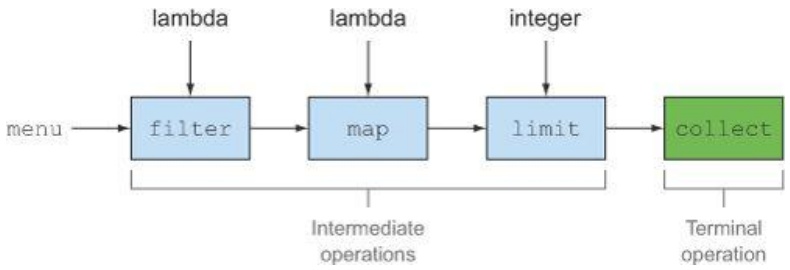
- **Types of stream operations**
  - **Intermediate Operations** (or lazy operations) - takes elements from an input stream and transforms the elements to produce an output stream
  - **Terminal Operations** (or eager operations) - takes inputs from a stream and produces the result

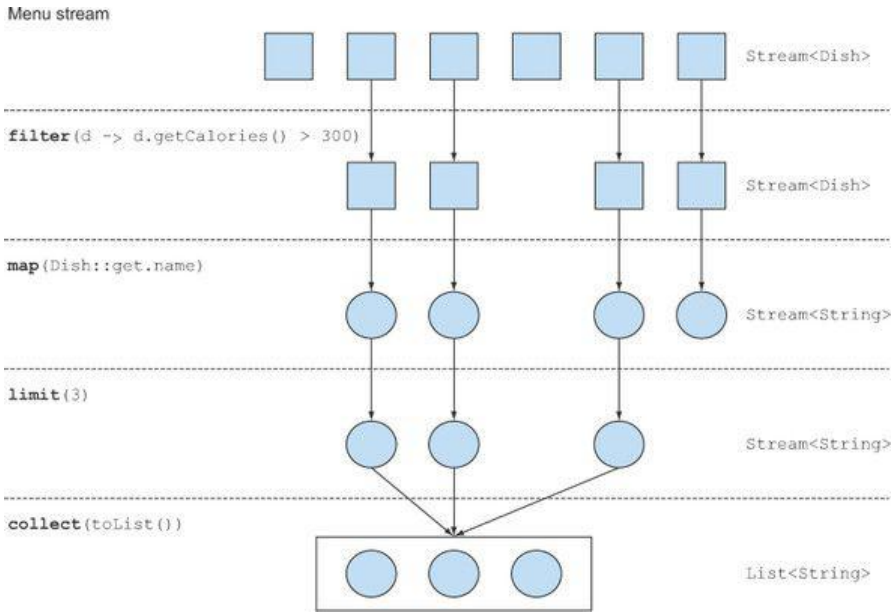
A stream is inherently lazy until you call a terminal operation on it!

Stream Operations

- **Example**

```
import static java.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames = menu.stream()
    .filter(d-> d.getCalories() > 300)
    .map(Dish::getName)
    .limit(3)
    .collect(toList());
System.out.println(threeHighCaloricDishNames);
```





Stream Operations

- **Intermediate Operations**
  - Return another stream; Allows operations to be connected as a query
  - Don't perform any processing until a terminal operation is invoked on the stream pipeline => they're lazy operations (stream traversal does not begin until the terminal operation of the pipeline is executed)
  - Reason: (i) Intermediate operations can usually be merged and processed into a single pass by the terminal operation or (ii) deployed on different processors in multi-processor map-reduce architectures
  - Examples: filter, map, distinct, sorted, peek, limit, skip, etc.
- **Terminal Operations**
  - Produce non-stream (a primitive value, a Collection or no value at all) result from a stream pipeline
  - They are preceded by intermediate operations on streams
  - Examples: forEach, forEachOrdered, reduce, collect, count, toList, toArray, min, max, anyMatch, allMatch, findAny, findFirst, etc.

Stream Operations

- **Terminal Operations - forEach**
  - Performs an action on each element of a stream considered as Consumer
  - May traverse the stream to produce a result or side effect

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

  - Examples

```
menu.stream().forEach(System.out::println);
```

Terminal Operations - sum, max, min, average, etc.

```
menu.stream()
    .map(Dish::getName)
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

```
final List<String> friends = Arrays.asList("Ion", "Vasile",
    "Liliana", "Sandu", "Maria", "Robert");
// Total number of characters in all names: " +

friends.stream()
          .mapToInt(name -> name.length())
          .sum());
```

Stream Operations

- **Terminal Operations - reduce**
  - Used when it is necessary to reduce a stream to a single value such as calculation the max, min, sum, product, etc.

```
T reduce(T identity, BinaryOperator<T> accumulator);
```

    - **identity** is the initial value (if missing is considered as zero)
    - **accumulator** is a **BinaryOperator** functional interface for combining two values
    - **BinaryOperator** represents an operation upon two operands of the same type, producing a result of the same type as the operands.
  - Show explicitly how the stream data is reduced
  - Example

```
int sumCalories = menu.stream()
    . map(Dish::getCalories)
    . reduce(0, (c1, c2) -> c1 + c2);
// The sum of all dishes calories is "
```

In each reduce iteration, c1 is the intermediate value of sumCalories while c2 is the new value in the stream

Stream Operations

- **Terminal Operations – collect() method**
  - Is a special case of reduction operation called mutable reduction operation
  - It returns mutable result container such as List, Set or Map as indicated by the supplied Collector
  - Example

```
import static java.util.Comparator.comparing;
import static util.stream.Collectors.toList;
   …
List<String> lowCaloricDishesName = menu.stream()
        .filter(d -> d.getCalories() < 400)
        .sorted(comparing(Dish::getCalories))
        .map(Dish::getName)
        .collect(Collectors.toList());
```

Stream Operations

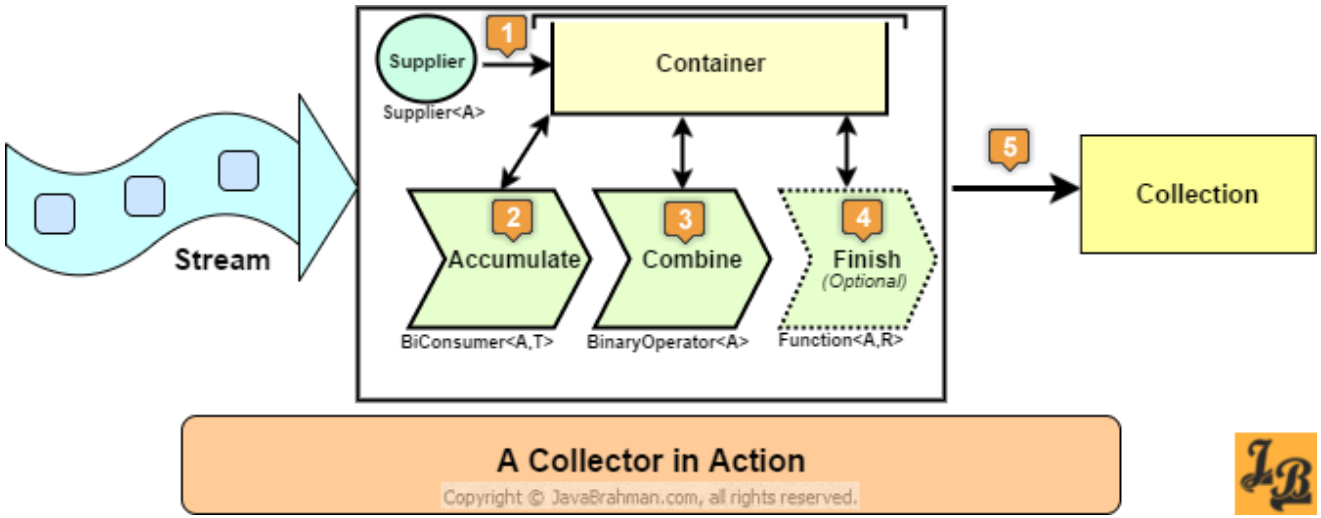
- **Terminal Operations – Collector interface**

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```

  - How Collector interface members are used by 4 components of a Collector
    - Supplier (FI Supplier) provides empty instance of type A to begin the accumulation of elements
    - Accumulator (FI BiConsumer) uses an instance of A to collect T
    - Combiner (FI BinaryOperator) combines two partial accumulated results of type A to produce a combined instance of A
    - Finisher maps A to R using a mapping function.

Stream Operations

- **Terminal Operations – Collector interface**



source: javabrahmsn.com

Supplier provides the mutable empty result container (instance of a Collection or Map) to hold the collected elements

Accumulator adds individual stream elements into the result container.

Combiner combines two partial results returned by two separate groups of accumulations done in parallel.

Optional Finisher put the processed elements in a desired form.

Finish Result The final collected elements are returned by the Collection in the result

Stream Operations

- **Terminal Operations – Collectors class**
  - Static methods that perform common reduction operations
    - accumulating elements into Collection (toList, toSet, toMap, toCollection),
    - min, max, average, sum of elements etc.
  - All the methods of Collectors class return Collector type which will be supplied to collect() method as an argument
  - Examples

```
String dishNamesJoined =
    menu.stream().map(Dish::getName).collect(Collectors.joining(", "));
```

```
int highestCalorie = menu.stream()
    .map(Dish::getCalorie).collect(Collectors.maxBy(Comparator.naturalOrder()));
```

Lazy and eager evaluation

- **Lazy evaluation**
  - delaying of the evaluation of an operation until it is needed
- **Eager evaluation**
  - an operation is executed when is encountered

```
IntUnaryOperator sampleMap =  num -> {
       System.out.println("number: " + num);
       return num;
    };
    Random random = new Random();
    IntStream randomStream = random
              .ints()
              .limit(5)
              .map(sampleMap)
              .sorted();
     System.out.println("AAAAAA");
    randomStream.forEach(System.out::println);
```

Output

```
AAAAAA
number: -1922978310
number: -387859319
number: -1632629133
number: -1079488471
number: 21584484
-1922978310
-1632629133
-1079488471
-387859319
21584484
```

The map method is not executed when the stream is declared, but when the forEach method is used. This is lazy evaluation – the expression is not evaluated until it is needed

Lazy and eager evaluation

- **Example**

```
List<String> names =
   menu.stream()
   .filter (d -> { System.out.println("filtering" + d.getName()); return d.getCalories() > 300;} )
   .map ( d -> {System.out.println("mapping" + d.getName()); return d.getName();})
   .limit(3)
   .collect(toList());
System.out.println(names);
```

Output:

filtering pork mapping pork filtering beef mapping beef filtering chicken mapping chicken
[pork, beef, chicken]

filter and map are two separate operations but for optimization they were merged into the same pass (loop fusion technique)

Creating Streams

- **Streams from Values**
  - Stream interface defines two static methods to create sequential stream from values
    - <T> Stream<T> of(T t)
    - <T> Stream<T> of(T...values)
- Examples

```
// Ex a. Creates a stream with one string elements
Stream<String> stream = Stream.of("Hello");
// Ex b. Creates a stream with four strings
Stream<String> stream = Stream.of(“Ion", “Vasile", “Sandu", “Nicolae");
// Ex c. Compute the sum of the squares of all odd integers in the list

int sum = Stream.of(1, 2, 3, 4, 5)
   .filter(n -> n % 2 == 1)
   .map(n -> n * n)
   .reduce(0, Integer::sum);
```

Creating Streams

- **Streams from Functions**
  - Stream interface contains two static methods to generate an infinite stream:

```
// iterate() - creates a sequential ordered stream
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

    - » The seed is the first element of the stream
    - » The second element is generated by applying the function to the first element, etc.

```
// generate() - creates a sequential unordered stream
static <T> Stream<T> generate(Supplier<T> s)
```

  - The stream interfaces for primitive values IntStream, LongStream, and DoubleStream also contain iterate() and generate() static methods that take parameters specific to their primitive types

Creating Streams

- **Streams from Functions – Examples**

```
public class PrimeUtil {
  private long lastPrime = 0L;

  // Calc prime after last generated
  public long next() {
    lastPrime = next(lastPrime);
    return lastPrime;
  }
  // Calc prime after specified  nmb
  public static long next(long after) {
    long counter = after;
    // loop until find the next prime
    while (!isPrime(++counter));
       return counter;
  }

public static boolean isPrime(long num) {
        if (num < 1) return false;
    if (num == 2 || num == 1) return true;
    if (num % 2 == 0) return false;
    for (int i = 3; i * i < num; i += 2)
        if (num % i == 0) return false;
    return true;
  }
}
```

```
// Ex 1 - creates an infinite stream of prime numbers
// and prints the first five prime numbers on the
// standard output: 2, 3, 5, 7, 11
Stream.iterate(2L, PrimeUtil::next)
  .limit(5)
  .forEach(System.out::println);
```

```
// Ex 2 – Alternative way
Stream.iterate(2L, n -> n + 1)
  .filter(PrimeUtil::isPrime)
  .limit(5)
  .forEach(System.out::println);
```

```
// Ex 3 – generate 5 random nmbs
// between 0.0 and 1.0
Stream.generate(Math::random)
  .limit(5)
  .forEach(System.out::println);
```

Creating Streams

- **Streams from Collections**

```
// Create a sequential stream from the set
Stream<String> sequentialStream = names.stream();

// Create a parallel stream from the set
Stream<String> parallelStream = names.parallelStream();
```

- Streams from files
  - Stream related I/O operations in Java 8 packages java.io and java.nio.file
  - Examples
    - Read text from a file as a stream of strings in which each element represents one line of text from the file
    - Obtaining the list of entries in a directory as a stream of Path

Streams and concurrent processing

- Concurrent processing
  - All stream operations execute either sequentially (default) or in parallel
  - To execute concurrently, a parallel stream must be created

- **Classical iterative Java loops are serial in nature; difficult to incorporate concurrent behavior**
- **Java threads for parallel execution using thread pools; drawbacks: possible data corruption and deadlock situations**
- **For parallel streams execution will ideally produce the same results as if executed serially but will execute faster**

Streams and concurrent processing

- Example of serial execution

```
Stream.of(“Ion”, “Vasile”, “Sandu”, “Ana”)
      .forEach(System.out::println);
```

Output serial (ordered elements):
Ion
Vasile
Sandu
Ana

- Example of parallel execution

```
Stream.of(“Ion”, “Vasile”, “Sandu”, “Ana”)
      .parallel()
      .forEach(System.out::println);
```

Output (possible) parallel
Sandu
Ion
Vasile
Ana

Streams and concurrent processing

- Factors to be considered when using parallel streams
  - **Non-inference**
    - During stream processing, its data source must not be modified
  - **Stateless operations**
    - Lambda expression whose outcome might vary during its execution is called stateful
    - As the stream's operations are executed, the results can differ each time
  - **Side effects**
    - A stream operation can affect other parts of a program (to be avoided if possible)
  - **Ordering**
    - The ordering of elements produced by a parallel stream may be important. If so, care must be taken to address the ordering issue

Streams and concurrent processing

- **Non-inference**
  - Occurs when the stream's data source is modified, during stream processing
    - Problem with non-concurrent data sources
  - There is always the possibility that some other thread may be accessing the data source
    - Race conditions , inaccurate results / exceptions
  - Examples

```
List<Integer> hours = new ArrayList (Arrays.asList(32, 40, 54, 23, 35, 48, 40, 45));
Stream<Integer> hoursStream = hours.parallelStream();
int totalHours = hoursStream
      .map(h -> {
           int amount = h*30;
           if(amount>1200) {hours.add(h+10);}
           return amount; })
      .reduce(0, (r, s) -> r + s);
```

ConcurrentModificationException is thrown!!!

Streams and concurrent processing

- **Non-inference**
  - How to avoid this problem?
    - Use CopyOnWriteArrayList class (allows concurrent modifications of the list)
    - CopyOnWriteArrayList - a thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array (Java documentation)
  - Examples

```
CopyOnWriteArrayList <Integer> concurrentHours =
      new CopyOnWriteArrayList (Arrays.asList(32, 40, 54, 23, 35, 48, 40, 45));
Stream<Integer> hoursStream = hours.parallelStream();
int totalHours = hoursStream
      .map(h -> {
         int amount = h*30;
         if(amount>1200) {concurrentHours.add(h+10);}
         return amount; })
     .reduce(0, (r, s) -> r + s);
System.out.println(totalHours);
```

Streams and concurrent processing

- **Stateless and side effects operations**
  - A stateless operation will not be affected by external programs
  - Operations should not modify other data elements of a program
  - If the operation has side effects => unforeseen consequences
    - In the example, a stream is used to add overtime hours to a separate list (overtimeList)
  - In FP, operating on global variables (side effects) should be avoided
    - In the example, the array list is also not thread safe => concurrent modification of the list may produce errors

```
List<Integer> overtimeList = new ArrayList<>();
List<Integer> hours = new ArrayList(Arrays.asList(32, 40, 54, 23, 35,
    48, 40, 45));
hours.parallelStream()
     .filter(s -> s > 40)
     .forEach(s -> overtimeList.add(s));
     for (Integer hour : overtimeList) {
        System.out.print(hour + " ");
    }
```

```
// The correct approach:
overtimeList =  hours
   .parallelStream()
   .filter(s -> s > 40)
  .collect(Collectors.toList());
```

Streams and concurrent processing

- **Ordering**
  - The ordering of stream elements can be important
  - When a stream is parallelized a stream, the order of the processed elements is affected
  - Example - sort the hours that are greater than 40 using a parallel stream

Possible (not sorted) output: 40, `32, 23, 54, 35, 48, 40, 45`
The order will vary with each execution because each parallel stream sorted its elements, but when the streams merge, they are not sorted.

To solve the problem, use the forEachOrdered method, which forces the stream to process the stream elements in the encountered order. However, this method can spoil the efficiency gained from parallel streams

```
hours.parallelStream()
    .filter(s -> s > 40)
    .sorted()
    .forEach(h -> System.out.print(h + " "));
 System.out.println();
```

```
hours.parallelStream()
   .filter(s -> s > 40)
   .sorted()
   .forEachOrdered(h -> System.out.print(h + " "));
System.out.println();
```