Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Streams

Definition

- Stream - sequence of data elements
  - Supports sequential and parallel aggregate operations
- Aggregate 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
- Streams support
  - Database-like operations
  - Common operations from functional programming languages to manipulate data, such as:
    - Filter,
    - Map,
    - Reduce
    - Find,
    - Match,
    - Sort, etc.
- Stream operations can be executed either sequentially or in parallel

Definition

- Stream is not Collection
  - Collections focus on storage of data elements for efficient access
  - Streams focus on aggregate computations on data elements from a data source that cold be collection;
  - Streams are consuming data from collections, arrays or I/O resources
  - Another view: *streams are smart iterators over collections*

Definition

- 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*
    - A pipeline of operations can be viewed as database-like query on the data source

Example

```
// Dish – an immutable class
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) );
```

[source: Urma]

Main features

Java 7 vs. Java 8

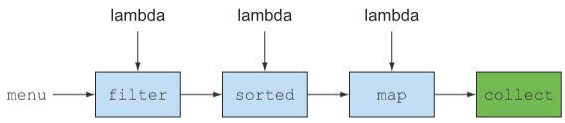
Java 7

```
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(Dish1, Dish2) {
    // 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));
```

- To use a multicore architecture and execute the code in parallel, one line of code needs to be changed:
  menu.stream()
- Should be:
  menu.parallelStream()
- Chaining stream operations to form a stream pipeline



[Source: Urma]

6

Main features

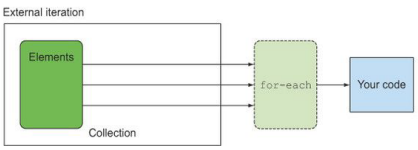
- Streams have no storage
  - A collection must store in memory (internal or external) all its elements
  - A stream has no storage; it does not store elements
  - A stream pulls (on-demand) elements from a data and passes them to a pipeline of operations for processing
- Streams can represent a sequence of infinite elements
  - A stream pulls its elements from a data source that can be a collection, a function that generates data, an I/O channel, etc.
- The design of streams is based on internal iteration
- Streams are designed to support functional programming
  - Stream operations don't modify the source data
  - Similar to FP you specify what operations should be performed on the data elements using the built-in methods provided by Streams API – typically by passing a lambda expression to those methods thus customizing the behavior of those operations
- 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

Internal and External 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 process 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



External Iteration
Source: [Urma]

External Iteration (for each statemen)

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

External Iteration (iterator object)

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

Main features

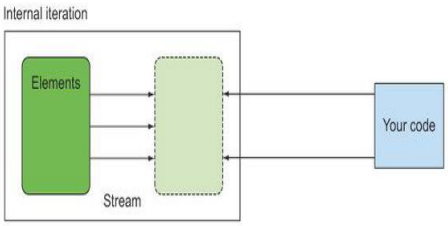
Internal and External Iteration

Internal Iteration

- Uses streams
- The iteration is achieved internally by the streams

- Example

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



Internal Iteration (stream like)
Source: [Urma]

Main features

Parallel Processing

Parallel stream 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

- Example

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

Stream Operations

Intermediate and Terminal Operations

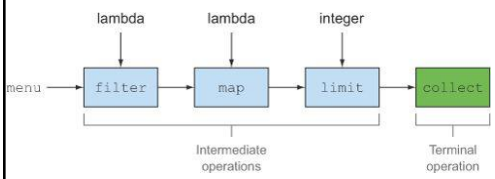
- Intermediate Operations (or lazy operations)
- Terminal Operations (or eager operations)
- A stream is inherently lazy until you call a terminal operation on it
- An intermediate operation on a stream produces another stream
- Each intermediate operation takes elements from an input stream and transforms the elements to produce an output stream.
- The terminal operation takes inputs from a stream and produces the result

Stream Operations

Intermediate and Terminal Operations

Example

```
import static.util.stream.Collectors.toList;
List<String> threeHighCaloricDishNames =
                                menu.stream()
  .filter(d-> d.getCalories() > 400)
  .map(Dish::getName)
  .limit(3)
  .collect(toList());
System.out.println(threeHighCaloricDishNames);
```



1. Get a stream from the list of dishes by calling the **stream** method on **menu**. The data source is the list of dishes (the menu) and it provides a sequence of elements to the stream

2. Apply a series of data processing operations on the stream: **filter, map, limit**, and **collect**. Filter, map and limit operations return another stream => they can be connected to form a pipeline, which can be viewed as a query on the source

3. At the end, **collect** operation starts processing the pipeline to return a result (it returns something different than a stream - here, a **List**)

Note. No result is produced, and indeed no element from menu is even selected, until **collect** is invoked. You can think of it as if the method invocations in the chain are queued up until collect is called (no memory associate with the streams)

Stream Operations

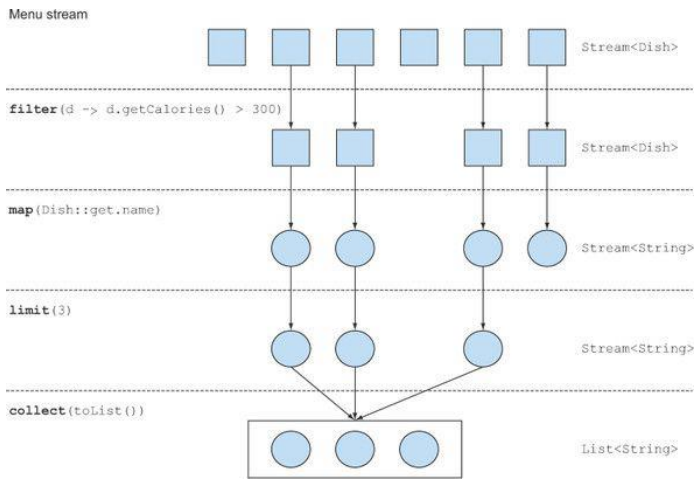
Intermediate and Terminal Operations

Operations description

- **filter** -takes a lambda (`d -> d.getCalories() > 400`) to define the filter
- **map** -takes a lambda, i.e. MR (`Dish::getName`) to transform an element into another one or to extract information.
  Note. `Dish::getName` is equivalent to lambda `d->d.getName()`
- **limit** - truncates a stream to contain no more than a given number of elements
- **collect** - converts a stream into another form (in this case into a list).

Stream Operations

Intermediate and Terminal Operations



Stream Operations

Intermediate and Terminal Operations

Intermediate Operations

- Return another stream as the return type
- Allows operations to be connected to form a query
- Intermediate operations don't perform any processing until a terminal operation is invoked on the stream pipeline—they're lazy.
- Reason: Intermediate operations can usually be merged and processed into a single pass by the terminal operation
- To understand what's happening in the stream pipeline (see on next slide) modify the code so each lambda also prints the current dish it's processing (useful learning and debugging technique)

Stream Operations

Intermediate and Terminal Operations

```
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() > 300);})
  .limit(3)
  .collect(toList());
System.out.println(names);
```

The output

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

Note.

1. See the optimizations due to the lazy nature of the streams
2. filter and map are two separate operations but for optimization they were merged into the same pass (*loop fusion* technique*)*

Stream Operations

Intermediate and Terminal Operations

Terminal Operations

- Terminal operations produce a result from a stream pipeline
- As a result they produce any non-stream value such as a List, an Integer, or even void.
- For example, in the following pipeline, **forEach** is a terminal operation that returns void and applies a lambda to each dish in the source

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

- Passing **System.out.println** to **forEach** asks it to print every Dish in the stream created from **menu**

Stream Operations

Declarative programming

- Stream oriented programming using lambda expressions - declarative style of programming
- **Declarative programming** style is very different than the imperative approach (the step by step way)
- 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

Creating Streams

- Streams from values
- Empty streams
- Streams from functions
- Streams from arrays
- Streams from collections
- Streams from files
- Streams from other sources

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

```
import java.util.stream.Stream;
...
int sum = Stream.of(1, 2, 3, 4, 5)
   .filter(n -> n % 2 == 1)
   .map(n -> n * n)
   .reduce(0, Integer::sum);
System.out.println("Sum = " + sum);
```

Creating Streams

Empty Streams

- Empty stream with no elements
- The Stream interface contains an **empty()** - static method to create an empty sequential stream

// Creates an empty stream of strings

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

- The IntStream, LongStream, and DoubleStream Functional Interfaces also contain an empty() static method to create an empty stream of primitive types.

// Creates an empty stream of integers

```
IntStream numbers = IntStream.empty();
```


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

- Examples for IntInterface:

IntStream iterate(int seed, IntUnaryOperator f)
IntStream generate(IntSupplier s)

- **Note.** An **infinite stream -** a stream with a data source *capable* of generating (on demand) infinite number of elements

Creating Streams

Streams from Functions

```
// PrimeUtil.java
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 number) {
    if (number <= 1) { return false; }
    if (number == 2) { return true; }
    if (number % 2 == 0) { return false; }
    long maxDivisor = (long) Math.sqrt(number);
    for (int counter = 3; counter <= maxDivisor;
          counter += 2) {
      if (number % counter == 0) { return false;}
    }
    return true;
  }
} // end class
```


Creating Streams

Streams from Functions – **iterate** examples

```
// 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 – Skips the first 100 prime numbers; Generates: 547, 557, 56, 569, 571
Stream.iterate(2L, PrimeUtil::next)
  .skip(100)
  .limit(5)
  .forEach(System.out::println);
```

Note. *forEach(Consumer<? Super T>, action)* as a terminal operation defined by Stream interface; *limit(long maxSize)* is defined by Stream interface

Creating Streams

Streams from Functions – **generate** examples

```
// Ex 1 – generate 5 random nmbs between 0.0 and 1.0
Stream.generate(Math::random)
  .limit(5)
  .forEach(System.out::println);

// Ex 2 – A PrimeUtil object acts as Supplier;
// next() remembers the last generated prime number
Stream.generate(new PrimeUtil()::next)
  .skip(100)
  .limit(5)
  .forEach(System.out::println);

// Ex 3 - Print five random integers
Stream.generate(new Random()::nextInt)
  .limit(5)
  .forEach(System.out::println);

// Ex 4 - Generate infinite stream of repeated values
IntStream zeroes = IntStream.generate(() -> 0);
```


Creating Streams

Streams from Collections

- Suppose a set **names** of type String

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

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

Creating Streams

Streams from Files

- New 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
  - Getting a stream of JarEntry from a JarFile
  - Obtaining the list of entries in a directory as a stream of Path

List of the most used Streams API operations

Operation	Type	Description
Distinct	Intermediate	Returns a stream consisting of the distinct elements of this stream. Elements e1 and e2 are considered equal if `e1.equals(e2)` returns true.
filter	Intermediate	Returns a stream consisting of the elements of this stream that match the specified predicate.
flatMap	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. The function produces a stream for each input element and the output streams are flattened. Performs one-to-many mapping.
limit	Intermediate	Returns a stream consisting of the elements of this stream, truncated to be no longer than the specified size.
map	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. Performs one-to-one mapping.
peek	Intermediate	Returns a stream whose elements consist of this stream. It applies the specified action as it consumes elements of this stream. It is mainly used for debugging purposes.
skip	Intermediate	Discards the first n elements of the stream and returns the remaining stream. If this stream contains fewer than n elements, an empty stream is returned.
sorted	Intermediate	Returns a stream consisting of the elements of this stream, sorted according to natural order or the specified `Comparator`. For an ordered stream, the sort is stable.
allMatch	Terminal	Returns true if all elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
anyMatch	Terminal	Returns true if any element in the stream matches the specified predicate, false otherwise. Returns false if the stream is empty.
findAny	Terminal	Returns any element from the stream. An empty `Optional` object is for an empty stream.
findFirst	Terminal	Returns the first element of the stream. For an ordered stream, it returns the first element in the encounter order; for an unordered stream, it returns any element.
noneMatch	Terminal	Returns true if no elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
forEach	Terminal	Applies an action for each element in the stream.
reduce	Terminal	Applies a reduction operation to computes a single value from the stream.