Functional programming, Streams, Lambda expressions in Java 8

Programming paradigms

- Imperative
  - uses statements that change a program's state
  - an imperative program consists of commands for the computer to perform
  - focuses on describing how a program operates
- Declarative
  - expresses the logic of a computation without describing its control flow
- Functional (subtype of declerative programming)
  - treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data

Functional programming

- Focusing and saying exactly just **what to do, not how** :
  - Let the dirty work to be done by *the “hidden” architecture* (it will be probably an optimized way), behind the executors.
  - Bonus: These executing process is going to improve, so your code will be quicker without touching it, you just need to update the specified language environment
- Programming is going to be more like creating an art with functional programming :)
- Immutability (transparent & clear functions)
- Attempts to minimize or eliminate side effects

Functional programming in Java 8

- How can we iterating over collection items?
    - For or while loops (as we did in Java 7)
    - Foreach for clean code
        - (using the iterator of the collection -> no NPE or index out of bound exception)
    - Iterable.foreach(lambda) (new in java 8)
    - Streams (new in java 8)

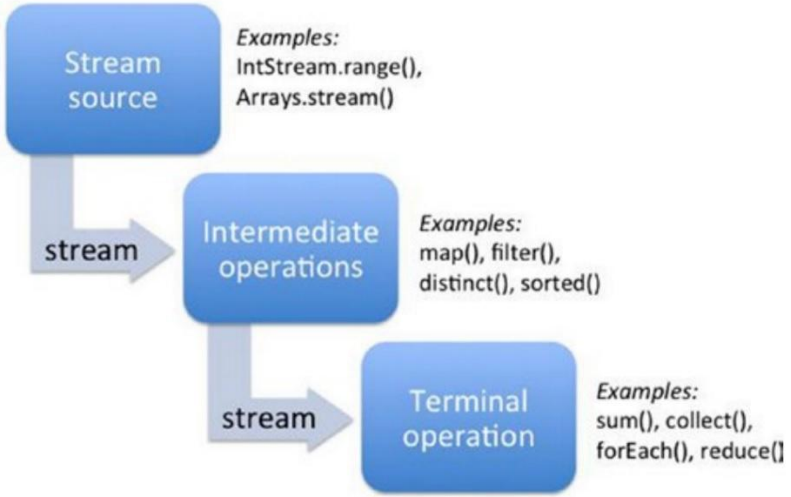
Streams + Lambdas in Java8

- Pros:
  - Compact
  - Bright structure, readability (focusing just on the logic)
  - Easily scalable (paralleling easily)
  - Easy to maintain
- Cons:
  - For small collections working with streams will cost more and will be slower than an old iteration
  - Difficult to test

What can streams built up from?

- Source +Terminal and intermediate elements of the stream pipe called operations

- Good to know about streams:
    - The source collection still remains the original (the result of the stream have to be collected and given to a variable)
    - Streams can be used just one time



- Operations (pipe-elements) executes the logic, which can be written in a:
  - Static method
  - Predicate<?> (methods with boolean return type)
  - Consumer<?> (void methods)
  - Supplier<?> (source)
  - Function<?,?> (any method can be assigned to a variable with this type)
- These parts can be written as:
  - Lambda (for small inline code, or in case of complex and not convenient)
  - Method reference (common usage, except in case of complex and not convenient)
- The operation can return:
  - Stream
  - Optional object (for example Optional<Integer>)
  - Primitive value or an object

Operations, the elements of the stream pipeline

- Intermediate operations
  - Filter
  - Map
  - Sort & sorting with comparator
  - Distinct
  - Peek(consumer) -> just for debugging purposes
  - Limit

- Common terminal operations
  - Collect
  - toArray
  - Foreach (+ Consumer)
  - Reduce
  - Sum
  - Count
  - Min,Max (+ Comparator)
  - FindFirst, findAny
  - anyMatch, allMatch, noneMatch

Imperative vs. Functional
Separation of Concerns

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```

```
List<String> errors =
    Files.lines(Paths.get(fileName))
         .filter(l -> l.startsWith("ERROR")
         .limit(40)
         .collect(toList());
```

Q&A

Thank you for your attention!

Useful links:

https://github.com/OCP-JavaSE7-StudyProjects/Java8Sandbox

http://overapi.com/?twitterID=nixCraft (Java + Git)

http://nngszegedanduszegedcollaboration.github.io/

https://msdn.microsoft.com/en-us/library/bb669144.aspx
https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.jpg