CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Streams - 1

Why streams? and not Lists?

- Streams have more convenient methods than Lists
  - forEach, filter, map, reduce, min, sorted, distinct, limit, etc.
- Streams have cool properties that Lists lack
  - Making streams more powerful, faster, and more memory efficient than Lists
  - The three coolest properties
    - Lazy evaluation
    - Automatic parallelization
    - Infinite (unbounded) streams
- Streams do not store data
  - They are just programmatic wrappers around existing data sources
  - Usually Lists or arrays, but later we will see you can have a function as a data source (Stream.generate), and the function is called each time that you need a Stream entry.

Not to confuse with I/OStreams

- **I/O streams**
  - Input streams: low-level data structures for reading from socket or file or other input source.
    - InputStream, ObjectInputStream, FileInputStream, ByteArrayInputStream, etc. Introduced in early Java versions.
  - Output streams: low-level data structures for sending data to socket or file.
    - OutputStream, ObjectOutputStream, FileOutputStream, ByteArrayOutputStream, etc.
- **Java 8 Stream interface**
  - Stream<T> (e.g., Stream<String>): High-level wrapper around arrays, Lists, and other data sources. Introduced in Java 8.
  - IntStream, DoubleStream, ... : Specializations of Java 8 streams for numbers.

Streams

- Wrappers around data sources such as arrays or lists.
- Support many convenient and high-performance operations expressed succinctly with lambdas, executed sequentially or in parallel.
- Quick preview

```
Employee firstRich =
  Stream.of(idArray).map(EmployeeUtils::findById)
                    .filter(e -> e != null)
                    .filter(e -> e.getSalary() > 500_000)
                    .findFirst()
                    .orElse(null));
```

Example: What does this codedo?

- Given very large file of words of various lengths in mixed case with possible repeats, create sorted uppercase file of n-letter words

```
List<String> words = Files.lines(Paths.get(inputFileName))
                          .filter(s -> s.length() == n)
                          .map(String::toUpperCase)
                          .distinct()
                          .sorted()
                          .collect(Collectors.toList());
Files.write(Paths.get(outputFileName), words,
            Charset.defaultCharset());
```

Characteristics of Streams

- **Not data structures**
  - Streams have *no* storage; they carry values from a source through a pipeline of operations.
    - They also never modify the underlying data structure (e.g., the List or array that the Stream wraps)
- **Designed for lambdas**
  - All Stream operations take lambdas as arguments
- **Do not support indexed access**
  - You can ask for the first element, but not the second or third or last element
  - But, see next bullet
- **Can easily be output as Lists or arrays**
  - Simple syntax to build a List or array from a Stream

Characteristics

- **Lazy**
  - Most Stream operations are postponed until it is known how much data is eventually needed
    - E.g., if you do a 10-second-per-item operation on a 100-element stream, then select the first entry, it takes 10 seconds, not 1000 seconds
- **Parallelizable**
  - If you designate a Stream as parallel, then operations on it will automatically be done in parallel, without having to write explicit fork/join or threading code
- **Can be unbounded**
  - Unlike with collections, you can designate a generator function, and clients can consume entries as long as they want, with values being generated on the fly

Making Streams: Overview

- Streams are not collections: they do not manage their own data. Instead, they are wrappers around existing data structures.
  - When you make or transform a Stream, it does not copy the underlying data. Instead, it just builds a pipeline of operations. How many times that pipeline will be invoked depends on what you later do with the stream (find the first element, skip some elements, see if all elements match a Predicate, etc.)

- 3 most common ways of making streams

```
someList.stream()
Stream.of(arrayOfObjects)
Stream.of(val1, val2, ...)
```

[not array of primitives!]

Making Streams: Examples

From Lists

```
List<String> words = …;
    words.stream().map(…).filter(…).other(…);
List<Employee> workers = …;
workers.stream().map(…).filter(…).other(…);
```

Making Streams: More Options

From List (and other collections)

- someList.stream(), someOtherCollection.stream()

From object array

- Stream.of(someArray), Arrays.stream(someArray)

From individual values

- Stream.of(val1, val2, …)

From a function

- Stream.generate, Stream.iterate

From a file

- Files.lines(somePath)

From a StreamBuilder

- someBuilder.build()

From String

- String.chars, Stream.of(someString.split(...))

From another Stream

- distinct, filter, limit, map, sorted, skip

Turning Streams into Pre-Java-8 Data Structures

- **List (most common)**
  - someStream.collect(Collectors.toList())
- **Array (less common)**
  - someStream.toArray(EntryType[]::new)
- **Note**
  - You normally do this only at the end, after you have done all the cool Stream operations. E.g.:

```
List<SomeType> values = someStream.map(...).filter(...).map(...).filter(...).collect(...);
```

Outputting Streams: Examples

- **Outputting as Lists**

```
List<String> wordList =
    someStream.map(someLambdaThatReturnsString)
              .filter(someTestOnStrings)
              .collect(Collectors.toList());
List<Employee> workerList =
   someStream.map(someLambda).collect(Collectors.toList());
```

- **Outputting as Arrays**

```
String[] wordArray =
    someStream.map(someLambdaThatReturnsString)
              .filter(someTestOnStrings)
              .toArray(String[]::new);
Employee[] workerList =
   someStream.map(someLambda).toArray(Employee[]::new);
```

Stream Methods

- You wrap a Stream around an array or List (or even a file). Then, you can do operations on each element (forEach), make a new Stream by transforming each element (map), remove elements that don't match some criterion (filter), etc.
- **Core methods**
  - forEach, map, filter, findFirst, findAny, collect, toArray
- **Other methods**
  - reduce, collect, min, max, sum, sorted, distinct, limit, skip, noneMatch, allMatch, anyMatch, count

Core Stream Methods

forEach(Consumer)

- `employees.forEach(e -> e.setSalary(e.getSalary() * 11/10))`

map(Function)

- `ids.map(EmployeeUtils::findEmployeeById)`

filter(Predicate)

- `employees.filter(e -> e.getSalary() > 500000)`

findFirst()

- `employees.filter(…).findFirst().orElse(defaultValue)`

toArray(ResultType[]::new)

- `Employee[] empArray = employees.toArray(Employee[]::new);`

collect(Collectors.toList())

- ```
  List<Employee> empList =
    employees.collect(Collectors.toList());
  ```
  ```

forEach

- **Calling a Lambda on Each Element of a Stream**
- Easy way to loop over Stream elements
  - There are also forEach methods directly in List (from Iterable), Map, etc.
- You supply a function (as a lambda) to forEach, and that function is called on each element of the Stream
  - More precisely, you supply a Consumer to forEach, and each element of the Stream is passed to that Consumer's accept method. But, few people think of it in these low-level terms.
- Quick examples
  - Print each element

```
Stream.of(someArray).forEach(System.out::println);
```

  - Clear all text fields

```
fieldList.stream().forEach(field -> field.setText(""));
```

forEach vs for Loops

for

```
for(Employee e: empList) {
  e.setSalary(e.getSalary() * 11/10);
}
```

forEach

```
empList().stream().forEach(e -> e.setSalary(e.getSalary() * 11/10));
```

- **Advantages of forEach**
  - Minor: designed for lambdas
    - Marginally more succinct
  - Minor: reusable
    - You can save the lambda and use it again (see example)
  - Major: can be made parallel with minimal effort
    - someStream.parallel().forEach(someLambda);

What You CANNOT do with forEach

- **Loop twice**
  - forEach is a "terminal operation", which means that it consumes the elements of the Stream. So, this is illegal:

```
someStream.forEach(element -> doOneThing(element));
someStream.forEach(element -> doAnotherThing(element));
```

  - But, of course, you can combine both operations into a single lambda
  - Also, you can use "peek" instead of forEach, and then loop twice
- **Change values of surrounding local variables**
  - Illegal attempt to calculate total yearly payroll:

```
double total = 0;
employeeList.stream().forEach(e -> total += e.getSalary());
```

  - But, we will see good way of doing this with "map" and "reduce".
  - In fact, this idea is so common that DoubleStream has builtin "sum" method
- **Break out of the loop early**
  - You cannot use "break" or "return" to terminate looping

map

- **Transforming a Stream by Passing Each Element through a Function**
- Produces a new Stream that is the result of applying a Function to each element of original Stream

- **Quick examples**
  - Array of squares

```
Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 };
Double[] squares = Stream.of(nums).
                        map(n -> n * n).toArray(Double[]::new);
```

  - List of Employees with given IDs

```
Integer[] ids = { 1, 2, 4, 8 };      // Integer[], not int[]
List<Employee> matchingEmployees =
        Stream.of(ids).map(EmployeeUtils::findById)
                      .collect(Collectors.toList());
```

map Example

- Helper Function

```
private static void printStream(Stream s, String message) {
  System.out.printf("%s: %s.%n",
                    message, s.collect(Collectors.toList()));
}
```

It is also common to first do
"import static java.util.stream.Collectors.*;"
and then to use toList() instead of Collectors.toList().

- Code:

```
List<Double> nums = Arrays.asList(1.0, 2.0, 3.0, 4.0, 5.0);
printStream(nums.stream(), "Original nums");
printStream(nums.stream().map(n -> n * n), "Squares");
printStream(nums.stream().map(n -> n * n).map(Math::sqrt),
            "Square roots of the squares");
```

- Output:

```
Original nums: [1.0, 2.0, 3.0, 4.0, 5.0].
Squares: [1.0, 4.0, 9.0, 16.0, 25.0].
Square roots of the squares: [1.0, 2.0, 3.0, 4.0, 5.0].
```

filter

- **Keeping Only the Elements that Pass a Predicate**
- Produces a new Stream that contain only the elements of the original Stream that pass a given test (Predicate)

- Quick Examples

Even numbers

```
Integer[] nums = { 1, 2, 3, 4, 5, 6 };
Integer[] evens = Stream.of(nums).filter(n -> n%2 == 0)
                                 .toArray(Integer[]::new);
```

Even numbers greater than 3

```
Integer[] evens = Stream.of(nums).filter(n -> n%2 == 0)
                                 .filter(n -> n>3)
                                 .toArray(Integer[]::new);
```

Example

- Code

```
Integer[] nums = { 1, 2, 3, 4, 5, 6 };
printStream(Stream.of(nums), "Original nums");
printStream(Stream.of(nums).filter(n -> n%2 == 0), "Even nums");
printStream(Stream.of(nums).filter(n -> n>3), "Nums > 3");
printStream(Stream.of(nums).filter(n -> n%2 == 0).filter(n -> n>3),  "Even nums > 3");
```

- Results

```
Original nums: [1, 2, 3, 4, 5, 6].
Even nums: [2, 4, 6].
Nums > 3: [4, 5, 6].
Even nums > 3: [4, 6].
```

findFirst

- **Returning the First Element of a Stream while Short-Circuiting Earlier Operations**
- Returns an *Optional* for the first entry in the Stream. Since Streams are often results of filtering, there might not be a first entry, so the Optional could be empty.
  - There is also a similar findAny method, which might be faster for parallel Streams.
- findFirst is faster than it looks when paired with map or filter. More details in section on lazy evaluation, but idea is that map and filter know to stop after a single entry is found.
- Examples:
  - When you know for certain that there is at least one entry
    - someStream.map(...).findFirst().get()
  - When unsure if there are entries or not (more common)
    - someStream.filter(...).findFirst().orElse(otherValue)

Optional Class

- Optional either stores a T or stores nothing. Useful for methods that may or may not find a value. New in Java 8.
  - The value of findFirst of Stream<Blah> is an Optional<Blah>
- Syntax
  - Making an Optional (usually done behind the scenes by builtin methods)
    - Optional<Blah> value = Optional.of(someBlah);
    - Optional<Blah> value = Optional.empty(); //Missing val
- Most common operations on an Optional (often done by your code)
  - value.get() – returns value if present or throws exception
  - value.orElse(other) – returns value if present or returns other
  - value.orElseGet(Supplier) – returns value if present or calls function
  - value.isPresent() – returns true if value is present

Example: Setup code

```
public class EmployeeSamples {

  private static List<Employee> GOOGLERS = Arrays.asList(
    new Employee("Larry", "Page", 1, 9999999),
    new Employee("Sergey", "Brin", 2, 8888888),
    new Employee("Eric", "Schmidt", 3, 7777777),
    new Employee("Nikesh", "Arora", 4, 6666666),
    new Employee("David", "Drummond", 5, 5555555),
    new Employee("Patrick", "Pichette", 6, 4444444),
    new Employee("Susan", "Wojcicki", 7, 3333333),
    new Employee("Peter", "Norvig", 8, 900000),
    new Employee("Jeffrey", "Dean", 9, 800000),
    new Employee("Sanjay", "Ghemawat", 10, 700000),
    new Employee("Gilad", "Bracha", 11, 600000) );

  public static List<Employee> getGooglers() {
    return(GOOGLERS);
  }

  ...
```

Example: Setup code

```
private static final List<Employee> SAMPLE_EMPLOYEES = Arrays.asList(
  new Employee("Harry", "Hacker", 1, 234567),
  new Employee("Polly", "Programmer", 2, 333333),
  new Employee("Cody", "Coder", 8, 199999),
  new Employee("Devon", "Developer", 11, 175000),
  new Employee("Desiree", "Designer", 14, 212000),
  new Employee("Archie", "Architect", 16, 144444),
  new Employee("Tammy", "Tester", 19, 166777),
  new Employee("Sammy", "Sales", 21, 45000),
  new Employee("Larry", "Lawyer", 22, 33000),
  new Employee("Amy", "Accountant", 25, 85000) );

public static List<Employee> getSampleEmployees() {
  return(SAMPLE_EMPLOYEES);
}

...
```

Example

Note: Employee with ID 8 is first match

```
Integer[] ids = { 16, 8, 4, ...};  // 10,000 entries
System.out.printf("First Googler with salary over $500K: %s%n",
                  Stream.of(ids).map(EmployeeSamples::findGoogler)
                                .filter(e -> e != null)
                                .filter(e -> e.getSalary() > 500_000)
                                .findFirst()
                                .orElse(null));
```

- How many times is:
  - findGoogler called? 2
  - The null check performed? 2
  - getSalary called? 1
- What if there were 10,000,000 ids instead of 10,000 ids?

Lazy Evaluation

"I'm not lazy, I'm just highly motivated to do nothing."

- Author Unknown

(but suspected Java Stream designer)

Overview

- Streams defer doing most operations until you actually need the results
- **Result**
  - Operations that appear to traverse Stream multiple times actually traverse it only once
  - Due to "short-circuit" methods, operations that appear to traverse entire stream can stop much earlier.
    - **stream.map(someOp).filter(someTest).findFirst().get()**
      - Does the map and filter operations *one element at a time* (first a map, then a filter on element 1, then map and filter on element 2, etc.). Continues only until first match on the filter test.
    - **stream.map(...).filter(...).filter(...).allMatch(someTest)**
      - Does the one map, two filter, and one allMatch test *one element at a time*. The first time it gets false for the allMatch test, it stops.

Method Types: Overview

- **Intermediate methods**
  - These are methods that produce other Streams. These methods don't get processed until there is some terminal method called.
- **Terminal methods**
  - After one of these methods is invoked, the Stream is considered consumed and no more operations can be performed on it.
    - These methods can do a side-effect (forEach) or produce a value (findFirst)
- **Short-circuit methods**
  - These methods cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.
    - Short-circuit methods can be intermediate (limit, skip) or terminal (findFirst, allMatch)
  - E.g., this example only filters until it finds *first* match:

    Stream.of(someArray).filter(e -> someTest(e)).findFirst().orElse(default)

Method Types: Listing by Categories

- **Intermediate methods**
  - map (and related mapToInt, flatMap, etc.), filter, distinct, sorted, peek, limit, skip, parallel, sequential, unordered
- **Terminal methods**
  - forEach, forEachOrdered, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator
- **Short-circuit methods**
  - anyMatch, allMatch, noneMatch, findFirst, findAny, limit, skip

Example

```
Stream.of(idArray).map(EmployeeUtils::findById)
                  .filter(e -> e != null)
                  .filter(e -> e.getSalary() > 500_000)
                  .findFirst()
                  .orElse(null));
```

- **Apparent behavior**
  - findById on all, check all for null, call getSalary on all non-null (& compare to $500K) on all remaining, find first, return it or null
- **Actual behavior**
  - findById on first, check it for null, if pass, call getSalary, if salary > $500K, return and done. Repeat for second, etc. Return null if you get to the end and never got match.

Checking Order of Operations

- Let's define the following to check:

```
Function<Integer,Employee> findGoogler =
  n -> { System.out.println("Finding Googler with ID " + n);
         return(EmployeeSamples.findGoogler(n));
       };
Predicate<Employee> checkForNull =
  e -> { System.out.println("Checking for null");
         return(e != null);
       };
Predicate<Employee> checkSalary =
  e -> { System.out.println("Checking if salary > $500K");
         return(e.getSalary() > 500_000);
       };
```

```
Integer[] ids = { 16, 8, 4, 2, 1 };
System.out.printf("First Googler with salary over $500K: %s%n",
                  Stream.of(ids).map(findGoogler)
                                .filter(checkForNull)
                                .filter(checkSalary)
                                .findFirst()
                                .orElse(null));
```

Order of Operations and Short-Circuiting

- Code

```
Integer[] ids = { 16, 8, 4, 2, 1 };
System.out.printf("First Googler with salary over $500K: %s%n",
                  Stream.of(ids).map(findGoogler)
                                .filter(checkForNull)
                                .filter(checkSalary)
                                .findFirst()
                                .orElse(null));
```

- Results

```
Finding Googler with ID 16
Checking for null
Finding Googler with ID 8
Checking for null
Checking if salary > $500K
First Googler with salary over $500K: Peter Norvig [Employee#8 $900,000]
```

If you thought of Streams as collections, you would think:
- It would first call findGoogler on all 5 ids, resulting in 5 Employees
- It would then call checkForNull on all 5 Employees
- It would then call checkSalary on all remaining Employees
- It would then get the first one (or null, if no Employees)

Instead, it builds a pipeline that, for each element in turn, calls findGoogler, then checks that same element for null, then if non-null, checks the salary of that same element, and if it exists, returns it.

Summary

- Make a Stream
  - someList.stream(), Stream.of(objectArray), Stream.of(e1, e2...)
- Output from a Stream
  - stream.collect(Collectors.toList())
  - stream.toArray(Blah[]::new)
- forEach [void output]
  - employeeStream.forEach(e -> e.setPay(e.getPay() * 1.1))
- map [outputs a Stream]
  - numStream.map(Math::sqrt)
- filter [outputs a Stream]
  - employeeStream.filter(e -> e.getSalary() > 500_000)
- findFirst [outputs an Optional]
  - stream.findFirst().get(), stream.findFirst().orElse(other)