Medii de programare

Lecture 6

Outline

- Functional programming (cont.)
- Concurrency

Optionals

`java.util.Optional`

- a container for a value which may be null or not
- prevents *NullPointerException*
- is *not* a functional interface

e05

Streams

`java.util.stream.Stream`

e06

Laziness, processing order

```
Stream.of("d2",  "a2", "b1", "b3", "c")
    .filter(s    -> {
      System.out.println("filter:     " + s);
      return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

What is the output of the following code sequence?

```
Stream.of("d2",  "a2", "b1", "b3", "c")
     .map(s  -> {
       System.out.println("map:    " + s);
       return s.toUpperCase();
     })
     .anyMatch(s -> {
       System.out.println("anyMatch:   " + s);
       return s.startsWith("A");
     });
```

Operation order matters

Reusing streams
problem:

1. `Stream<String>   stream = Stream.of("d2", "a2", "b1", "b3", "c")`
   `.filter(s  -> s.startsWith("a"));`
2. `stream.anyMatch(s    -> true); // ok`
3. `stream.noneMatch(s   -> true); //` **`exception`**

solution:

1. `Supplier<Stream<String>> streamSupplier = ()-> Stream.of("d2", "a2", "b1", "b3", "c")`
   `.filter(s    -> s.startsWith("a"));`
2. `streamSupplier.get().anyMatch(s   -> true); // ok`
3. `streamSupplier.get().noneMatch(s  -> true); // ok`

Collect, reduce
e07

Concurrency