INTRODUCING “JAVA 8”

Brief Introduction to new features in Java 8

Stream API

- Introduction to Streams API
- Streams v/s Lists
- Stream methods:
  - forEach
  - map
  - filter
  - findFirst
  - collect
- Streams and lambda expression
- Lazy evaluations

Streams

- Wrappers around collections that support many convenient and high performance operations expressed succintly with lambda.
- Example:

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

Streams v/s Lists

- Streams have more convenient methods
  - forEach, filter, map, reduce etc
- Streams have cool properties
  - Lazy evaluation
  - Automatic parallelization
  - infinite (unbounded) streams

features

- Not data structures
- Designed for lambda
- Do not support indexed access
- Can easily output as arrays or lists

Making Streams

From Individual Values	Stream.of(val1,val2...)
From an Array	Stream.of(someArray)
From List Or Collection	List1.stream() OR coll1.stream()

Stream methods

- forEach
  - Easy way to loop over Stream elements
  - You supply a lambda to forEach, and that lambda is called on each element of the Stream
  - eg.
    ```
    Stream.of(someArray).forEach(System.out::println);
    ```
- map
  - Produces a new Stream that is the result of applying a Function to each element of original Stream
  - eg,
    Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 }; Double[] squares = Stream.of(nums).map(n -> n * n).toArray(Double[]::new);

Stream Methods

- filter
  - Produces a new Stream that contain only the elements of the original Stream that pass a given test
  - eg.

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

- findFirst
  - 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.
  - eg.

```
someStream.map(…).findFirst().get()
```

Stream methods

- collect
  - Combined with methods in the Collectors class, you can build many data types out of Streams
  - eg.

```
Integer[] nums = {1,2,3,4,5};
List<Integer> k  =
Arrays.stream(nums).limit(3).collect(Collectors.
toList());
```

Streams with lambda expressions

- You supply a lambda to forEach, and that lambda is called on each element of the Stream
- eg.

```
Stream<Employee> employees =
getEmployees().stream(); employees.forEach(e ->
e.setSalary(e.getSalary() * 11/10));
```

- Similarly, can be used with other methods like map, filter etc.

Lazy evaluation

- Many 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 list, then select the first entry, it takes 10 seconds, not 1000 seconds