JAVA Streams (Java 8)

Amos Azaria

Dealing With Data

- There are basically two ways to deal with data:
  - In the database: anything related to data is done only in the database. We can use stored procedures to compute complicated queries on the data.
  - In code: the database only stores the data, any smart and complex logic is performed by the software (e.g. java). When dealing with "disposable data" may not even require any database.

Motivation for Java Streams

- Suppose we have an array containing data, how would we compute a value on this data (e.g., the average/sum/minimum, or sum of all items > 5)?
- Loops: In c, we would iterate using the indexes (and compute whatever we need).

```
double sum = 0;
for (int i = 0; i < LENGTH; i++)
{
    sum += a[i];
}
```

Motivation for Java Streams- Cont.

- In Java, we gain one level of abstraction by using "for each in" syntax (which is still a loop).

```
double sum = 0;
for (double t : a)
{
    sum += t;
}
```

- Writing a loop is not non-natural, what we really want is to compute something on all the data without dealing with the overhead of a loop.

Motivation for Java Streams- Cont.

- As we will see, Java Streams also allows us to use parallel execution.

```
double sum = Arrays.stream(a).sum();

double sum = Arrays.stream(a).filter(a->a>5).sum();
```

Lambda Expressions (Lambda Calculus)

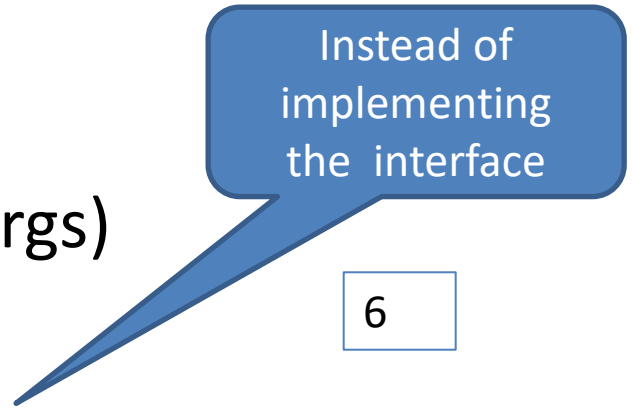
- Using Streams relies heavily on the use of lambda expressions.
- Lambda expressions are actually anonymous functions. Introduced in Java 8, but long present in functional programming languages.

- Syntax: (argument-list) -> {body}
  - E.g.: (x,y) -> return x+y
-

Lambda Expressions in Java

- Full example:

```
interface IntOps
{
    int unaryOp(int a);
}
public static void main(String[] args)
{
    IntOps increment = (a -> a+1);
    System.out.println(increment.unaryOp(5));
}
```



Lambda Expressions: Function<,>

Get the **T** and return the **R**

- Using the Function<T,R> class, we can discard the IntOps Interface:

```
Function<Integer,Integer> increment = x -> x + 1;
System.out.println(increment.apply(5));
```

Functional Interface

```
6
```

Functions as Parameters

```
public static Function<Integer,Integer>
        getFun(Function<Integer,Integer> inputFun)
{
   return x -> inputFun.apply(x + 2);
}

public static void main(String[] args)
{
   System.out.println(getFun(x -> x*3).apply(4));
}
```

Callable

- We can also define a function that doesn't get any arguments. This is useful for suspended execution (e.g. undo in Command design pattern):

```
Callable<Integer> printAndReturn7 = () -> {int x = 7;
System.out.println(x);    return x; };
System.out.println(printAndReturn7.call());
```

```
7
7
```

Stream

- Stream is a Java class, but very rarely do we declare a variable of type stream.
- A Stream is a sequence of elements.
- These elements are obtained from arrays, collections (such as lists) or I/O resources.
- Stream operations usually return streams, which in turn can be further processed.
- Collectors can turn streams back into collections.

Data Processing (Not necessarily using a DBMS)

- Streams in Java 8, allow aggregated processing of data in a declarative way, similarly to SQL. E.g.:
  - filter
  - map
  - sort
  - reduce
  - find
  - Match
  - ….
- C# (.net) supports Linq, which is (a better implementation) of the same idea.

Stream Example

```
List<String> myList = Arrays.asList("yes", "no",
"hello", "goodbye", "none");
Stream s1 = myList.stream();
Stream s2 = s1.sorted();
List<String> sortedList =
(List<String>)(s2.collect(Collectors.toList()));
for (String item : sortedList)
{
   System.out.println(item);
}
```

Sorted can also get 'Comparator' parameter, e.g- S1.sorted(Comparator.reverseOrder())

```
goodbye
hello
no
none
yes
```

How We Would *Actually* Do It!

```
List<String> myList = Arrays.asList("yes", "no", "hello",
"goodbye", "none");
myList.stream().sorted().forEach((a)->System.out.println(a));

myList.stream().sorted().forEach(System.out::println);
```

Method Reference

```
goodbye
hello
no
none
yes
```

Another Example

- Suppose we have a classrooms Map from classroomsNames to a list of students that learn in that classroom: Map<String, List<Student>> classrooms
- An instance of type Student has a field: reqAccesability
- We want to get only the classes that have students that require accessibility.

```
List<String> roomsReqAccess = new LinkedList<>();
for (Map.Entry<String, List<Student>> classroom : classrooms.entrySet()){
   for (Student student : classroom.getValue()){
      if (student.reqAccessibility){
         roomsReqAccess.add(classroom.getKey());
         break;
      }
   }
}
```

Another Example

And with JavaStreams:

```
List<String> roomsReqAccess = classrooms.entrySet()
                    .stream()
                    .filter(a->a.getValue()
                            .stream()
                            .anyMatch(x->x.reqAccesability))
                    .collect(Collectors.toList(c->c.getKey()));
```

anyMatch() is a terminal operation and returns a Boolean value

Terminal operations: anyMatch, allMatch, noneMatch, findAny, findFirst, collect, reduce, forEach...

Parallel Execution

- When:
  - We don't care about the order of processing (usually the case).
  - Processing every item takes non-negligible amounts of time.
  - (Or) There are many items to process.
  - We have available threads.
- We may want to use parallelStream(), instead of stream().

Parallel execution: prime number example

```
private static boolean naivePrimeTest(long num)
{
    if (num <= 2)
        return true;
    if (num % 2 == 0)
        return false;
    for (long i = 3; i*i <= num; i += 2)
    {
        if (num % i == 0)
            return false;
    }
    return true;
}
```

Parallel execution: prime number example (cont.)

```
Random r = new Random();
List<Long> numberList = new LinkedList<>();
for (int i = 0; i < 1000; i++)
{
    numberList.add(r.nextLong());
}
long start = System.nanoTime();
long numberOfPrimes =
    numberList.parallelStream().filter(a -> naivePrimeTest(a)).count();
long end = System.nanoTime();
System.out.println("numberOfPrimes: " + numberOfPrimes + ".
time: " + ((end-start)*1E-9) + " seconds");
```

numberOfPrimes: 18.
time: 133.20204266 seconds

numberOfPrimes: 22.
time: 48.913557517 seconds

BigInteger.isProbablePrime()

- If you actually want to find prime numbers, you should use a *much* faster (probabilistic) method:

```
long numberOfPrimes = numberList.stream().filter(a ->
BigInteger.valueOf(a).isProbablePrime(100)).count();
```

$1 - \frac{1}{2^n}$

1000 numbers sequential: numberOfPrimes: 26. time: 0.188116916 seconds	100,000 numbers sequential: numberOfPrimes: 2502. time: 2.09736326 seconds
1000 numbers parallel: numberOfPrimes: 33. time: 0.18128167 seconds	100,000 numbers parallel: numberOfPrimes: 2552. time: 1.17280468 seconds

reduce() in Streams

- The reducer is the aggregator / combiner of the stream operations. Many reducers are built-in, e.g.:
  - count()
  - sum()
  - collect()
- It is guaranteed that the result of the reducer will be equal to a sequential combiner (even when using parallelStream, assuming the combiner works in order, e.g. Collectors.toList()).
- Some streams do not need a reducer, for example, if they end in …forEach(System.out::println);
- Java streams also supports using our own reducer.

reduce() in Streams (cont.)

- recude([identity], accumulator, [combiner])
  - accumulator: a lambda function with two inputs:
    1. The partial result of the output thus far
    2. The current element.

    This function outputs the partial result after accumulating current element.
  - combiner: a lambda function with two partial results as inputs returning the partial result of both combined.
  - identity: a starting element that doesn't alter the result, (e.g. 0 in sum, or 1 in multiplication). If we do not provide an identity parameter, we obtain an optional result (in case no items were processed).

reduce() in Streams (cont.)

If we don't specify a combiner, the accumulator serves also as the combiner

- E.g.:
  - …sum() is equivalent to: …reduce(0, (x,y)-> x+y)
  - …count() is equivalent to: …reduce(0, (x,t)-> x+1, (x,y) -> x+y)
  - Concatenate 7th char's of every string:
    - …reduce("", (x,t) -> x + t.charAt(6), (x,y) -> x + y);

Map-Reduce Example

- Suppose we have documents with the following structure:

```
{
    _id: ObjectId("50a8240b927d5d8b5891743c"),
    cust_id: "abc123",
    ord_date: new Date("Oct 04, 2012"),
    status: 'A',
    amount: 25,
    items: [ { sku: "mmm", qty: 5, price: 2.5 },
             { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

- We would like to get the total **amount** paid by each cust_id with status 'A'.

How Can we solve it in Java Streams?

- Assume we would have all the data in Java objects, can we obtain the query using Java Streams?
- orders.stream()

```
.filter(a->a.status.equals("A"))
.map(a-> a.amount)
.reduce(0, (x,y)-> x+y);
```

3249

Map() gets a stream and changes the elements according to the function it gets as parameter

Not what we want! We don't want just to sum all the quantities with status "A", we want to sum them per cust_id!

Solution …

```
Map<String,Double> totalPerCustomer = orders
    .stream()
    .filter(a->a.status.equals("A"))
    .collect(Collectors.groupingBy(
        a->a.cust_id,
        Collectors.reducing(0., x->x.amount, (x,y) -> x+y)
));
```

Grouping "key"

Identity

"mapping" on each item (what information actually interests us)

How to combine two partial results

{customer_a=1100, customer_b =1568, customer_c=845, customer_d=86}

Solution …

```
Map<String,Double> totalPerCustomer = orders
    .stream()
    .filter(a->a.status.equals("A"))
    .collect(Collectors.groupingBy(
          a->a.cust_id,
          Collectors.summingDouble(a->a.amount)
));
```

Grouping "key"

{customer_a=1100, customer_b =1568, customer_c=845, customer_d=86}