CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Streams – 3

Infinite (Unbounded On-the-Fly) Streams

- **Stream.generate(valueGenerator)**
  - Stream.generate lets you specify a Supplier. This Supplier is invoked each time the system needs a Stream element.
    - Powerful when Supplier maintains state, but then parallel version will give wrong answer
- **Stream.iterate(initialValue, valueTransformer)**
  - Stream.iterate lets you specify a seed and a UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) becomes third element, etc.
- **Usage**
  - The values are not calculated until they are needed
  - To avoid unterminated processing, you must eventually use a size-limiting operation like limit or findFirst (but not skip alone)
    - The point is not really that this is an "infinite" Stream, but that it is an unbounded "on the fly" Stream – one with no fixed size, where the values are calculated as you need them.

generate

- You supply a function (Supplier) to Stream.generate. Whenever the system needs stream elements, it invokes the function to get them.
- You must limit the Stream size.
  - Usually with limit or findFirst (or findAny for parallel streams). skip alone is not enough, since the size is still unbounded
- By using a real class instead of a lambda, the function can maintain state so that new values are based on any or all of the previous values
- Quick Example

```
List<Employee> emps =
  Stream.generate(() -> randomEmployee())
        .limit(someRuntimeValue)
        .collect(Collectors.toList());
```

Stateless generate Example: Random Numbers

- Code

```
Supplier<Double> random = Math::random;
System.out.println("2 Random numbers:");
Stream.generate(random).limit(2).forEach(System.out::println);
System.out.println("4 Random numbers:");
Stream.generate(random).limit(4).forEach(System.out::println);
```

- Results

```
2 Random numbers:  0.00608980775038892
0.2696067924164013
4 Random numbers:
0.7761651889987567
0.818313574113532
0.07824375091607816
0.7154788145391667
```

Stateful generate Example: Supplier Code

```
public class FibonacciMaker implements Supplier<Long> {
  private long previous = 0;
  private long current = 1;

  @Override
  public Long get() {
    long next = current + previous;
    previous = current;
    current = next;
    return(previous);
  }
}

public static Stream<Long> makeFibStream() {
  return(Stream.generate(new FibonacciMaker()));
}

public static Stream<Long> makeFibStream(int numFibs) {
  return(makeFibStream().limit(numFibs));
}

public static List<Long> makeFibList(int numFibs) {
  return(makeFibStream(numFibs).collect(Collectors.toList()));
}

public static Long[] makeFibArray(int numFibs) {
  return(makeFibStream(numFibs).toArray(Long[]::new));
}
```

Lambdas cannot define instance variables, so we use a regular class instead of a lambda to define the Supplier.

Example

- Main code

```
System.out.println("5 Fibonacci numbers:");
FibStream.makeFibStream(5).forEach(System.out::println);
System.out.println("25 Fibonacci numbers:");
FibStream.makeFibStream(25).forEach(System.out::println);
```

- Sample Output

```
5 Fibonacci numbers:
1
1
2
3
5
25 Fibonacci numbers:
1
1
...
75025
```

iterate

- You specify a seed value and a UnaryOperator f. The seed becomes the first element of the Stream, f(seed) becomes the second element, f(second) [i.e., f(f(seed))] becomes third element, etc.
- You must limit the Stream size.
  - Usually with limit. skip alone is not enough, since the size is still unbounded
- Will *not* yield the same result in parallel
- Quick Example:

```
List<Integer> powersOfTwo =
  Stream.iterate(1, n -> n * 2)
        .limit(…)
        .collect(Collectors.toList());
```

Simple Example: Twitter Messages

- Generate a series of Twittermessages
- **Approach**
  - Start with a very short String as the first message
  - Append exclamation points on the end
  - Continue to 140-character limit
- **Core Code**

```
Stream.iterate("Base Msg", msg -> msg + "Suffix")
      .limit(someCutoff)
```

- **Example**

```
System.out.println("14 Twitter messages:");
Stream.iterate("Big News!!", msg -> msg + "!!!!!!!!!!")
      .limit(14)
      .forEach(System.out::println);
```

Output:

```
Big News!!
Big News!!!!!!!!!!
Big News!!!!!!!!!!!!!!!!!!!!
Big News!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Consecutive Large Prime Numbers

- Generate a series of very large consecutive prime numbers (e.g., 100 or 150 digits or more)
- Large primes are used extensively in cryptography
- **Approach**
  - Start with a prime BigInteger as the seed
  - Supply a UnaryOperator that finds the first prime number higher than the given one
- **Core code**

```
Stream.iterate(Primes.findPrime(numDigits),
                    Primes::nextPrime)
      .limit(someCutoff)
```

Helper Method: nextPrime

- Generate a random odd BigInteger of the requested size, check if prime, keep adding 2 until you find a match.
- Why this is feasible
  - The BigInteger class has a builtin probabilistic algorithm (Miller-Rabin test) for determining if a number is prime without attempting to factor it. It is ultra-fast even for 100-digit or 200-digit numbers.
  - Technically, there is a ~$1/2^{100}$ chance that this falsely identifies a prime, but since 2^{100} is about the number of particles in the universe, that's not a very big risk

```
public static BigInteger nextPrime(BigInteger start) {
  if (isEven(start)) start = start.add(ONE);
  else start = start.add(TWO);
 if (start.isProbablePrime(ERR_VAL)) return(start);
  else return(nextPrime(start));
}
public static BigInteger findPrime(int numDigits) {
  if (numDigits < 1) numDigits = 1;
  return(nextPrime(randomNum(numDigits)));
}
```

Making Stream of Primes

```
public static Stream<BigInteger> makePrimeStream(int numDigits) {
  return(Stream.iterate(Primes.findPrime(numDigits), Primes::nextPrime));
}
public static Stream<BigInteger> makePrimeStream(int numDigits, int numPrimes) {
  return(makePrimeStream(numDigits).limit(numPrimes));
}
public static List<BigInteger> makePrimeList(int numDigits, int numPrimes) {
  return(makePrimeStream(numDigits, numPrimes).collect(Collectors.toList()));
}
public static BigInteger[] makePrimeArray(int numDigits, int numPrimes) {
  return(makePrimeStream(numDigits, numPrimes).toArray(BigInteger[]::new));
}
```

- Try it

```
System.out.println("10 100-digit primes:");
PrimeStream.makePrimeStream(100, 10).forEach(System.out::println);
```

```
10 100-digit primes:
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797635 3
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797664 7
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797666 3
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797668 9
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797723 3
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797785 9
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797788 9
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797798 9
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797803 1
348489448980592459903325950159955996105790357274387010597234543845855625353127126284846355286797810 3
```

collect

- Using methods in the Collectors class, you can output a Stream as many types
- Quick Examples
  - List (shown in previous section)
    - `anyStream.collect(toList())`
  - String
    - `stringStream.collect(joining(delimiter)).toString()`
  - Set
    - `anyStream.collect(toSet())`
  - Other collection
    - `anyStream.collect(toCollection(CollectionType::new))`
  - Map
    - `strm.collect(partitioningBy(...))`, `strm.collect(groupingBy(...))`

For brevity, the examples here assume you have done "import static java.util.stream.Collectors.*", so that toList() really means Collectors.toList()

partitoningBy: Building Maps

- You provide a Predicate. It builds a Map where true maps to a List of entries that passed the Predicate, and false maps to a List that failed the Predicate.
- Quick Example:

```
Map<Boolean,List<Employee>> oldTimersMap =
  employeeStream().collect(partitioningBy(e -> e.getEmployeeId() < 10));
```

- Now, oldTimersMap.get(true) returns a List<Employee> of employees whose ID's are less than 10, and oldTimersMap.get(false) returns a List<Employee> of everyone else.

partitoningBy: Example

- Code

```
Map<Boolean,List<Employee>> richTable =
  googlers.stream().collect(partitioningBy(e -> e.getSalary() > 1_000_000));
System.out.printf("Googlers with salaries over $1M: %s.%n", richTable.get(true));
System.out.printf("Destitute Googlers: %s.%n", richTable.get(false));
```

- Results

```
Googlers with salaries over $1M: [Larry Page [Employee#1 $9,999,999],
  Sergey Brin [Employee#2 $8,888,888], Eric Schmidt [Employee#3 $7,777,777],  Nikesh
  Arora [Employee#4 $6,666,666], David Drummond [Employee#5 $5,555,555],
  Patrick Pichette [Employee#6 $4,444,444], Susan Wojcicki [Employee#7 $3,333,333]].

Destitute Googlers: [Peter Norvig [Employee#8 $900,000],
  Jeffrey Dean [Employee#9 $800,000], Sanjay Ghemawat [Employee#10 $700,000],  Gilad
  Bracha [Employee#11 $600,000]].
```

groupingBy: Another Way of Building Maps

- You provide a Function. It builds a Map where each output value of the Function maps to a List of entries that gave that value.
  - E.g., if you supply Employee::getFirstName, it builds a Map where supplying a first name yields a List of employees that have that first name.
- Quick Example

```
Map<Department,List<Employee>> deptTable =
  employeeStream()
    .collect(Collectors.groupingBy(Employee::getDepartment));
```

- Now, deptTable.get(someDepartment) returns a List<Employee> of everyone in that department.

Example: groupBy Offices

```
private static Emp[] sampleEmps = {
  new Emp("Larry", "Page", "Mountain View"),  new Emp("Sergey", "Brin",
  "Mountain View"),  new Emp("Lindsay", "Hall", "New York"),
  new Emp("Hesky", "Fisher", "New York"),
  new Emp("Reto", "Strobl", "Zurich"),
  new Emp("Fork", "Guy", "Zurich"),
};
public static List<Emp> getSampleEmps() {
  return(Arrays.asList(sampleEmps));
}
Map<String,List<Emp>> officeTable =
  EmpSamples.getSampleEmps().stream()
          .collect(Collectors.groupingBy(Emp::getOffice));
System.out.printf("Emps in Mountain View: %s.%n", officeTable.get("Mountain View"));
System.out.printf("Emps in NY: %s.%n", officeTable.get("New York"));
System.out.printf("Emps in Zurich: %s.%n", officeTable.get("Zurich"));
```

Output:

```
Emps in Mountain View:
  [Larry Page [Mountain View], Sergey Brin [Mountain View]].
Emps in NY: [Lindsay Hall [New York], Hesky Fisher [New York]].
Emps in Zurich: [Reto Strobl [Zurich], Fork Guy [Zurich]].
```

Summary

- Reduction operations on Stream<T>
- Number Specialized Streams
- Parallel streams
  - anyStream.parallel().normalStreamOps(...)
    - For reduce, be sure there is no global data and that operator is associative
    - Test to verify answers are the same both ways, or (with doubles) at least close enough
    - Compare timing
- "Infinite" (really unbounded) streams
  - Stream.generate(someStatelessSupplier).limit(...)
  - Stream.generate(someStatefullSupplier).limit(...)
  - Stream.iterate(seedValue, operatorOnSeed).limit(...)
- Fancy uses of collect
  - You can build many collection types from streams