Streams API

Don't micro-manage me!

Objectives

- Extract streams from collections
- Know how to force a stream to produce data
- Describe the benefits of lazy streams
- Use the `filter`, `map`, `flatMap`, `peek`, and `forEach` operations on a stream
- Use Optional objects
- Perform reduction and collection operations
- Use primitive streams and convert between primitive and object streams
- Use parallel streams

Why Streams?

- Streams allow a more declarative approach to defining computations
  - Specify what to achieve, rather than how to achieve it
- The primary goals are
  - Easier to read code, partly because it's shorter, partly because it's more declarative
  - Automatic parallelization, allowing use of concurrent hardware (multi-core CPUs) without error prone hand-written threading and locking code

A Simple Example

```
List<Student> ls = Arrays.asList(
 new Student("Fred", 2.3F), new Student("Jo", 3.3F),
 new Student("Tara", 3.8F), new Student("Yi", 2.8F),
 new Student("Sal", 1.8F), new Student("Ben", 4.0F)
);

ls
   .stream()
   .filter(s->s.getGpa()>=3.0F)
   .forEach(s->System.out.println("> " + s));
```

A Simple Example

- Streams can be created from a number of sources, collections and files being common
- Streams are used in a fashion somewhat similar to Unix pipes; a sequence of operations are chained together in a “production line”
- Each operation in the production line may change the data, for example:
  - By filtering out items on some criterion
  - By selecting a particular data element

More Stream Foundations

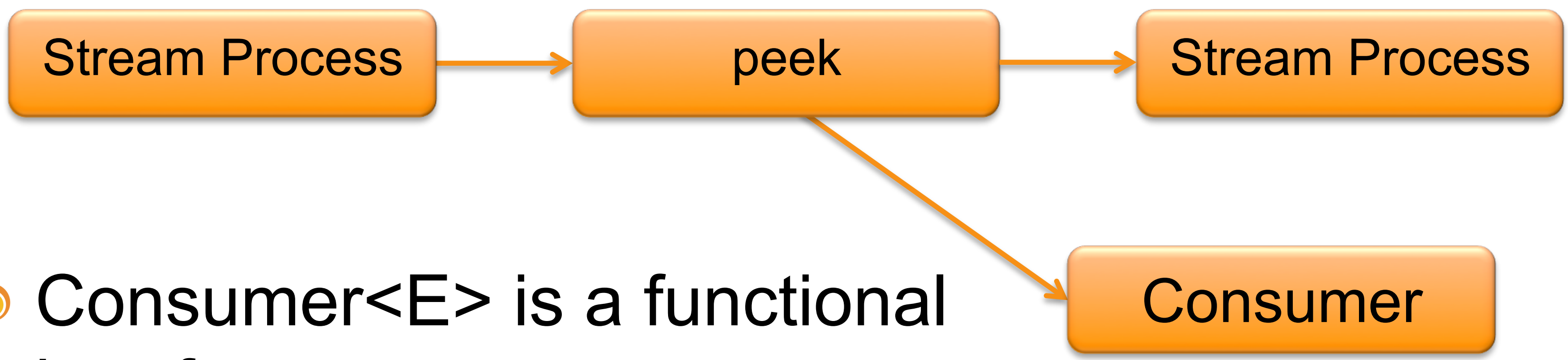
- Most streams are stream<E>, which works for objects
- Special streams are provided for int, long, and double; use these for numeric / computationally intensive work
- Tools are provided for converting between the different stream formats using a concept called “mapping”
  - Mapping has more general application than simply converting from one type of data to another, and will be discussed shortly

More Stream Foundations

- Streams are “lazy”, to improve computational efficiency
  - Imagine that data must be pulled down the pipe from the right hand end, rather than being pushed from their origin point
- Many stream methods return another stream; these are ***lazy*** methods
- Stream methods that return non-stream types will pull data through the stream
  - Ensure your stream ends with a non-lazy method!

Investigating Laziness

- The Stream.peek() method allows looking at data as it flows down a stream



- Consumer<E> is a functional interface

Investigating Lazyness

- This prints ***nothing***:

```
ls.stream()
  .peek(s->System.out.println("peeking > " + s))
  .filter(s->s.getGpa()>=3.0F);
```

- This shows every student once, and each matching item a second time:

```
ls
.stream()
  .peek(s->System.out.println("peeking > " + s))
  .filter(s->s.getGpa()>=3.0F)
  .forEach(s->System.out.println("> " + s));
```

More Non-Lazy Methods

- Also known as terminal methods

Method	Effect
`allMatch(Predicate)`	Reports is all stream items pass the test
`anyMatch(Predicate)`	Reports if any stream item passes the test
`collect`	Mutable reduce operation to arbitrary type
`count`	Returns the number of items in the stream
`findFirst`	Returns an Optional of the first stream item
`forEach(Consumer)`	Invoke the consumer for each stream item
`max(Comparator)`	Return Optional of the largest item
`min(Comparator)`	Return Optional of the smallest item
`noneMatch(Predicate)`	Report if zero items pass the test
`reduce`	Collect all the items to one of the same type
`toArray()`	Collect items into an array

Optional<T>

- Terminal stream methods might need to report "there wasn't any result"
- C.A.R Hoare describes `null` as his "billion dollar mistake" `Optional<T>` attempts to avoid the problems of `null`
- `Optional<T>` wraps an item that might be absent
  - `empty()` / `isPresent()`
  - `get()`
  - `ifPresent(Consumer<T>)`
  - `orElse(T other)` / `orElseThrow(Throwable t)`

Stream (Lazy) Methods

- Non-terminal methods return another Stream

Method	Effect
`distinct`	Removes duplicate items
`filter(Predicate)`	Removes non-matching items
`flatMap`	Flattens a stream of streams into one stream
`limit(long)`	Truncates the stream's length
`map(Function)`	Produces a stream of derived items
`peek(Consumer)`	Calls Consumer for each passing item
`skip(long)`	Removes items from start of stream
`sorted`	Orders items in the stream

Map Operations

- The map method converts the contents of the stream, based on a Function
- Data type of the stream can be changed at the same time

```
ls
   .stream()
   .map(s->s.getName())
   .distinct()
   .forEach(s->System.out.println(s));

   // or .map(Student::getName)
```

Flat Mapping

- If a stream contains items that may be treated as more than one thing, flatMap can produce a stream of single items

Fred		Carol
Jim	Tony	Hillary
Sheila	Jane	Humphrey

- flatMap produces

Fred	Jim	Sheila	Tony	Jane	Carol	Hillary	Humphrey

Flat Map Example

```
List<String[]> lsa = Arrays.asList(
  new String[] { "Fred", "Jim", "Sheila" },
  new String[] { "Tony", "Jane" },
  new String[] { "Carol", "Hillary", "Humphrey" }
);

lsa.stream()
  .flatMap(s->Arrays.stream(s))
  .forEach(s->System.out.println("> " + s));
```

A Stream of String[]

Turn each String[] into Stream<String>

Turn the Stream<Stream<String>> into a Stream<String>; put each String into the main stream in turn, losing the original grouping

Reduce Operations

- Reduction is a process of taking the elements of a stream and merging them into a single result
- The process should use only immutable data, this is one way to achieve reliable parallelism
- Each intermediate result is therefore a new data item
  - This might be wasteful of intermediate objects
- Basic reduction repeatedly takes two items and merges them to one
  - E.g. concatenating strings

Simple Reduce Example

```
Optional<String> res = ls.stream()
   .map(Student::getName)
   .distinct()
   .reduce((s,t)->s + ", " + t);
res.ifPresent(System.out::println);
```

The result is an Optional, because the stream might have been empty

Advanced Reduction

- More advanced reduction can give more control over the reduction process
- For a stream of type T, reduction can produce a potentially different type U, using the method:

```
U reduce(
    U identity,
    BiFunction<U,T,U> accumulator,
    BinaryOperator<U> combiner)
```

- Important note: The return values of accumulator and combiner ***must be new objects***
  - This supports concurrency, which will be examined later

Example Advanced Reduction

```
StringBuilder result = ls.stream()
   .map(Student::getName)
   .distinct()
   .reduce(
     new StringBuilder(),
     (s,t)->new StringBuilder(s + ", " + t),
     (s,t)->new StringBuilder(s + ", " + t));
System.out.println(result);
```

Identity

Combine StringBuilder and String to create a ***new*** StringBuilder

What's this "Identity"?
Why is result not an Optional?
What's "wrong" with the output?

Combine two StringBuilders to create a ***new*** StringBuilder

Reductions Using Identity

- Some stream reduction (and collection) operations collect using an "identity"
- This is typically treated as the first item in the sequence, and has some consequences:
  - The result is never nothing, so Optional is not required
  - If we try to separate the items on the list with, say a comma, then reductions using identity will cause many problems
- Really, an empty string is not an "identity", and complex reductions are for data processing, ***not string concatenation***

Collect Operations

- Collect operations allow mutable collector objects
  - Mutable collectors can be more efficient for large streams
  - Parallelism is supported, but using different internal techniques, also to be investigated later
- Two collect methods:

```
R collect(Collector<T,A,R> collector)
R collect(Supplier<R> supplier,
    BiConsumer<R,? super T> accumulator,
    BiConsumer<R,R> combiner)
```

Mutable Collection Containers

- Notice that the identity argument in a reduce operation was an object, but in the collect operation it's a Supplier of some kind
- In reduction, the identity constitutes is processed as an item of the stream
- In collection, the Supplier creates an empty collection object, and the items in the stream are mixed into this
  - The supplied object is continuously mutated by the collection process

Some Pre-Defined Collectors

- Class java.util.stream.Collectors provides many pre-defined utility collectors

Method	Behavior Of Resulting Collector
`counting`	Counts the elements in the stream
`groupingBy`	Various forms of classification operations
`joining`	Concatenates CharSequence items to String
`maxBy(Comparator)`	Find the “largest” item
`minBy(Comparator)`	Find the “smallest” item
`partitioningBy`	Group by pass / fail on the provided test
`reducing`	Reduce the items using a provided operation
`toList`	Collect the items into a List
`toMap`	Map items into key+value in a Map
`toSet`	Collect items into a set

Using Simple Pre-Defined Collectors

```
long count = ls.stream()
   .map(Student::getName)
   .distinct()
   .collect(Collectors.counting());
System.out.println("There are: "
     + count + " distinct student names");
```

Grouping Using Collectors

- `Collectors.groupingBy` creates a `Map` of data.
- Provide a function that extracts the key, data is added to a `List` in the value part

```
Map<String, List<Student>> map = ls.stream()
   .collect(Collectors.groupingBy(s->s.getName()));

map.forEach((k,v)->{
   System.out.println("Students named " + k);
   v.forEach(s->System.out.println(" > " + s));
});
```

Grouping And Downstream Collector

- The items added to the list in the groupingBy collector can be processed with a downstream collector

```
Map<String, Double> map = ls.stream()
  .collect(
    Collectors.groupingBy(Student::getName,
      Collectors.averagingDouble(Student::getGpa)
    )
  );
map.forEach((k, v) -> System.out.println(
  "Students named " + k + " averaged " + v)
);
```

This collector processes the items that would have been added to the list

Converting Primitive And Object Streams

- Recall that map operations convert data
- Recall that special streams exist for int, long, and double types (because generics cannot express primitives)
- Map-type operations also exist for converting between object-type and primitive-type streams

Converting Primitive And Object Streams

- Convert from object streams to primitives:

Method	Result
`mapToDouble`	Stream of double values
`mapToInt`	Stream of int values
`mapToLong`	Stream of long values

```
DoubleSummaryStatistics dss = ls.stream()
   .mapToDouble(Student::getGpa)
   .peek(d->System.out.println("gpa is " + d))
   .summaryStatistics();

System.out.println("Stats: " + dss);
```

Converting Primitive And Object Streams

- Convert from primitive streams to object streams:

Method	Result
`boxed`	Stream of object wrapper values (Integer etc.)
`mapToObj`	Stream of objects

Primitive Stream Terminal Operations

- Primitive streams support several unique terminal operations that perform mathematics

Method	Result
`average`	OptionalDouble defining arithmetic mean
`max`	Numerically largest
`min`	Numerically smallest
`sum`	number representing total of all items
`summaryStatistics`	Mean, count, max, min, and sum of the values

Primitive Streams And Collectors

- Note that the utility collectors provided in the Collectors class are generic; they operate on object streams, not primitive streams
  - Create your own collector, or map the primitive stream to an object stream (using autoboxing) so that the provided collectors can work

More Ways To Create Streams

- Static Stream methods returning a Stream

Method	Effect
`concat`	Join two streams
`empty`	Create an empty stream
`generate(Supplier)`	Ask the Supplier to create the itemss
`iterate(T seed, UnaryOperator<T>)`	Repeatedly invoke the operation to create each new item
`of(T …)`	Create a stream from the listed values

- Method `aBufferedReader.lines()`

Parallel Processing With Streams

- Streams can be run in parallel
- This is built into the VM and parallelization is handled automatically if permission is granted e.g.:

  ```
  aStream.parallel()...
  ```
- Can also explicitly require sequential operation:

  ```
  aStream.sequential()...
  ```
- ***An entire stream runs either sequentially, or in parallel***. If calls to parallel and sequential exist at different points in the stream code, the last one wins for the whole stream

Parallel Processing With Streams

- Stream processing code for use in parallel processing should follow some simple rules
  - Don’t depend on the order of items, this cannot be guaranteed
  - Don’t allow reduce behaviors to modify any data, they must create new objects at every step
  - Don’t use data that’s not part of the stream, it might be subject to concurrent access
    - If you don’t protect it, your code will break
    - If you do protect it, you’ll probably damage performance

Lab Exercise

- Download a plain text book from the Gutenberg project http://www.gutenberg.org/
- Write a program that computes the “concordance” for the book
- A concordance is a word frequency table, indicating each word that is ever used in the book, and how many times it is used

Lab Exercise

- Examine the stream method `generate` in the Java API documentation
- Use this method, with other stream behaviors to write a method that prints a row of *n* asterisks where *n* is an `int` argument to the method

Lab Exercise

- Write a program that simulates the rolls of three dice. Simulate a single throw using:
  `ThreadLocalRandom.current().nextInt(1,7)`
- Three dice will show a face value between three and 18; count the frequency of each face value
- Arrange to display a bar chart showing the relative frequencies
- Run the program with 100,000 throws, time the execution
- Arrange to use parallel execution, and re-time the 100,000 throw execution