JAVA 8 STREAM API

Outline

- Stream Building Blocks
  - Java 8
  - Default Methods
  - Functional Interfaces
  - Lambda Expressions
  - Method References

Outline

- Characteristics of Streams
- Creating Streams
- Common Functional Interfaces Used
- Anatomy of the Stream pipeline
- Optional Class
- Common Stream API Methods Used
  - Examples
- Parallel Streams
- Unbounded (On the Fly) Streams
- What Could Streams Do For BMI
- References
- Questions?

Java 8

- Target Release Date: 03/18/14
- Introduces
  - Default Methods
  - Functional Interfaces
  - Lambda Expressions
  - Stream API and overall improvements to Collections to support Streams

Default Methods

- In Context of Support For Streams
  - Java 8 needed to add functionality to existing Collection interfaces to support Streams (stream(), forEach())

Default Methods

- Problem
  - Pre-Java 8 interfaces couldn't have method bodies.
  - The only way to add functionality to Interfaces was to declare additional methods which would be implemented in classes that implement the interface
  - It is impossible to add methods to an interface without breaking the existing implementation

Default Methods

- Solution
  - Default Methods!
  - Java 8 allows default methods to be added to interfaces with their full implementation
  - Classes which implement the interface don't have to have implementations of the default method
  - Allows the addition of functionality to interfaces while preserving backward compatibility

Default Methods

- Example

```
public interface A {
    default void foo(){
    System.out.println("Calling A.foo()");
}

public class Clazz implements A {}

Clazz clazz = new Clazz();
clazz.foo(); // Calling A.foo()
```

Functional Interfaces

- Interfaces with only one abstract method.
- With only one abstract method, these interfaces can be easily represented with lambda expressions
- Example

```
@FunctionalInterface
public interface SimpleFuncInterface {
    public void doWork();
}
```

Lambda expressions

- A more brief and clearly expressive way to implement functional interfaces
- Format: <Argument List> -> <Body>
- Example (Functional Interface)

```
public interface Predicate<T> {
        boolean test(T input);
}
```

- Example (Static Method)

```
public static <T> Collection<T> filter(Predicate<T> predicate,
        Collection<T> items) {
        Collection<T> result = new ArrayList<T>();
        for(T item: items) {
                if(predicate.test(item)) {
                        result.add(item);
                }
        }
}
```

- Example (Call with Lambda Expression)

```
Collection<Integer> myInts = asList(0,1,2,3,4,5,6,7,8,9);
Collection<Integer> onlyOdds = filter(n -> n % 2 != 0, myInts)
```

Method References

- Event more brief and clearly expressive way to implement functional interfaces
- Format: <Class or Instance>::<Method>
- Example (Functional Interface)

```
public interface IntPredicates {
    boolean isOdd(Integer n) { return n % 2 != 0; }
}
```

- Example (Call with Lambda Expression)

```
List<Integer> numbers = asList(1,2,3,4,5,6,7,8,9);
List<Integer> odds = filter(n -> IntPredicates.isOdd(n), numbers);
```

- Example (Call with Method Reference)

```
List<Integer> numbers = asList(1,2,3,4,5,6,7,8,9);
List<Integer> odds = filter(IntPredicates::isOdd,  numbers);
```

Characteristics of Streams

- Streams are not related to InputStreams, OutputStreams, etc.
- Streams are NOT data structures but are wrappers around Collection that carry values from a source through a pipeline of operations.
- Streams are more powerful, faster and more memory efficient than Lists
- Streams are designed for lambdas
- Streams can easily be output as arrays or lists
- Streams employ lazy evaluation
- Streams are parallelizable
- Streams can be “on-the-fly”

Creating Streams

- From individual values
  - Stream.of(val1, val2, …)
- From array
  - Stream.of(someArray)
  - Arrays.stream(someArray)
- From List (and other Collections)
  - someList.stream()
  - someOtherCollection.stream()

Common Functional Interfaces Used

- Predicate<T>
  - Represents a predicate (boolean-valued function) of one argument
  - Functional method is boolean Test(T t)
    - Evaluates this Predicate on the given input argument (T t)
    - Returns true if the input argument matches the predicate, otherwise false
- Supplier<T>
  - Represents a supplier of results
  - Functional method is T get()
    - Returns a result of type T

Common Functional Interfaces Used

- Function<T,R>
  - Represents a function that accepts one argument and produces a result
  - Functional method is R apply(T t)
    - Applies this function to the given argument (T t)
    - Returns the function result
- Consumer<T>
  - Represents an operation that accepts a single input and returns no result
  - Functional method is void accept(T t)
    - Performs this operation on the given argument (T t)

Common Functional Interfaces Used

- UnaryOperator<T>
  - Represents an operation on a single operands that produces a result of the same type as its operand
  - Functional method is R Function.apply(T t)
    - Applies this function to the given argument (T t)
    - Returns the function result

Common Functional Interfaces Used

- BiFunction<T,U,R>
  - Represents an operation that accepts two arguments and produces a result
  - Functional method is R apply(T t, U u)
    - Applies this function to the given arguments (T t, U u)
    - Returns the function result
- BinaryOperator<T>
  - Extends BiFunction<T, U, R>
  - Represents an operation upon two operands of the same type, producing a result of the same type as the operands
  - Functional method is R BiFunction.apply(T t, U u)
    - Applies this function to the given arguments (T t, U u) where R,T and U are of the same type
    - Returns the function result
- Comparator<T>
  - Compares its two arguments for order.
  - Functional method is int compareTo(T o1, T o2)
    - Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Anatomy of the Stream Pipeline

- A Stream is processed through a pipeline of operations
- A Stream starts with a source data structure
- Intermediate methods are performed on the Stream elements. These methods produce Streams and are not processed until the terminal method is called.
- The Stream is considered consumed when a terminal operation is invoked. No other operation can be performed on the Stream elements afterwards
- A Stream pipeline contains some short-circuit methods (which could be intermediate or terminal methods) that cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.

Anatomy of the Stream Pipeline

- Intermediate Methods

  map, filter, distinct, sorted, peek, limit,

  parallel
- Terminal Methods

  forEach, toArray, reduce, collect, min,

  max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator
- Short-circuit Methods

  anyMatch, allMatch, noneMatch, findFirst, findAny,limit

Optional<T> Class

- A container which may or may not contain a non-null value
- Common methods
  - isPresent() – returns true if value is present
  - Get() – returns value if present
  - orElse(T other) – returns value if present, or other
  - ifPresent(Consumer) – runs the lambda if value is present

Common Stream API Methods Used

- Void forEach(Consumer)
  - Easy way to loop over Stream elements
  - You supply a lambda for forEach and that lambda is called on each element of the Stream
  - Related peek method does the exact same thing, but returns the original Stream

Common Stream API Methods Used

- Void forEach(Consumer)
  - Example

    Employees.forEach(e -> e.setSalary(e.getSalary() * 11/10))

    Give all employees a 10% raise

Common Stream API Methods Used

- Void forEach(Consumer)
  - Vs. For Loops

```
List<Employee> employees = getEmployees();
for(Employee e: employees) {
    e.setSalary(e.getSalary() * 11/10);
}
```

  - Advantages of forEach
    - Designed for lambdas to be marginally more succinct
    - Lambdas are reusable
    - Can be made parallel with minimal effort

Common Stream API Methods Used

- Stream<T> map(Function)
  - Produces a new Stream that is the result of applying a Function to each element of original Stream
  - Example

    Ids.map(EmployeeUtils::findEmployeeById)

    Create a new Stream of Employee ids

Common Stream API Methods Used

- Stream<T> filter(Predicate)
  - Produces a new Stream that contains only the elements of the original Stream that pass a given test
  - Example

    employees.filter(e -> e.getSalary() > 100000)

    Produce a Stream of Employees with a high salary

Common Stream API Methods Used

- Optional<T> findFirst()
  - Returns an Optional for the first entry in the Stream
  - Example

    employees.filter(…).findFirst().orElse(Consultant)

    Get the first Employee entry that passes the filter

Common Stream API Methods Used

- Object[] toArray(Supplier)
  - Reads the Stream of elements into a an array
  - Example

    Employee[] empArray = employees.toArray(Employee[]::new);

    Create an array of Employees out of the Stream of Employees

Common Stream API Methods Used

- List<T> collect(Collectors.toList())
- Reads the Stream of elements into a List or any other collection
  - Example

    List<Employee> empList =
    employees.collect(Collectors.toList());

    Create a List of Employees out of the Stream of Employees

Common Stream API Methods Used

- List<T> collect(Collectors.toList())
  - partitioningBy
    - 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.
    - Example

```
Map<Boolean,List<Employee>> richTable =
googlers().collect
(partitioningBy(e -> e.getSalary() > 1000000));
```

  - groupingBy
    - 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.
    - Example

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

Common Stream API Methods Used

- T reduce(T identity, BinaryOperator)
- You start with a seed (identity) value, then combine this value with the first Entry in the Stream, combine the second entry of the Stream, etc.
  - Example

    Nums.stream().reduce(1, (n1,n2) -> n1*n2)

    Calculate the product of numbers
- IntStream (Stream on primative int] has build-in sum()
- Built-in Min, Max methods

Common Stream API Methods Used

- Stream<T> limit(long maxSize)
- Limit(n) returns a stream of the first n elements
  - Example

    someLongStream.limit(10)

    First 10 elements

Common Stream API Methods Used

- Stream<T> skip(long n)
- skip(n) returns a stream starting with element n
  - Example

    twentyElementStream.skip(5)

    Last 15 elements

Common Stream API Methods Used

- Stream<T> sorted(Comparator)
  - Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator
  - Example

    empStream.map(…).filter(…).limit(…)

    .sorted((e1, e2) -> e1.getSalary() - e2.getSalary())

    Employees sorted by salary

Common Stream API Methods Used

- Optional<T> min(Comparator)
  - Returns the minimum element in this Stream according to the Comparator
  - Example

```
Employee alphabeticallyFirst =
ids.stream().map(EmployeeSamples::findGoogler)
.min((e1, e2) ->
e1.getLastName()
.compareTo(e2.getLastName()))
.get();
```

Get Googler with earliest lastName

Common Stream API Methods Used

- Optional<T> max(Comparator)
  - Returns the minimum element in this Stream according to the Comparator
  - Example

```
Employee richest =
ids.stream().map(EmployeeSamples::findGoogler)
.max((e1, e2) -> e1.getSalary() -
e2.getSalary())
.get();
```

Get Richest Employee

Common Stream API Methods Used

- Stream<T> distinct()
  - Returns a stream consisting of the distinct elements of this stream
  - Example

```
List<Integer> ids2 =
Arrays.asList(9, 10, 9, 10, 9, 10);
List<Employee> emps4 =
ids2.stream().map(EmployeeSamples::findGoogler)
.distinct()
.collect(toList());
```

Get a list of distinct Employees

Common Stream API Methods Used

- Boolean anyMatch(Predicate), allMatch(Predicate), noneMatch(Predicate)
  - Returns true if Stream passes, false otherwise
  - Lazy Evaluation
    - anyMatch processes elements in the Stream one element at a time until it finds a match according to the Predicate and returns true if it found a match
    - allMatch processes elements in the Stream one element at a time until it fails a match according to the Predicate and returns false if an element failed the Predicate
    - noneMatch processes elements in the Stream one element at a time until it finds a match according to the Predicate and returns false if an element matches the Predicate
  - Example

    employeeStream.anyMatch(e -> e.getSalary() > 500000)

    Is there a rich Employee among all Employees?

Common Stream API Methods Used

- long count()
  - Returns the count of elements in the Stream
  - Example

    employeeStream.filter(somePredicate).count()

    How many Employees match the criteria?

Parallel Streams

- Helper Methods For Timing

```
private static void timingTest(Stream<Employee> testStream)
{
    long startTime = System.nanoTime();
    testStream.forEach(e -> doSlowOp());
    long endTime = System.nanoTime();
    System.out.printf(" %.3f seconds.%n",
    deltaSeconds(startTime, endTime));
}

private static double deltaSeconds(long startTime, long
endTime) {
    return((endTime - startTime) / 1000000000);
}
```

Parallel Streams

- Helper Method For Simulating Long Operation

```
void doSlowOp() {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException ie) {
        // Nothing to do here.
    }
}
```

Parallel Streams

- Main Code

```
System.out.print("Serial version [11 entries]:");
timingTest(googlers());
int numProcessorsOrCores =
Runtime.getRuntime().availableProcessors();
System.out.printf("Parallel version on %s-core machine:",
numProcessorsOrCores);
timingTest(googlers().parallel() );
```

Parallel Streams

- Results

  Serial version [11 entries]: 11.000 seconds.

  Parallel version on 4-core machine: 3.000 seconds.

(On The Fly) Streams

- Stream<T> generate(Supplier)
  - The method lets you specify a Supplier
  - This Supplier is invoked each time the system needs a Stream element
  - Example

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

- Stream<T> iterate(T seed, UnaryOperator<T> f)
  - The method lets you specify a seed and a UnaryOperator.
  - The seed becomes the first element of the Stream, f(seed) becomes the second element of the Stream, f(second) becomes the third element, etc.
  - Example

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

- The values are not calculated until they are needed
- To avoid unterminated processing, you must eventually use a size-limiting method
- This is less of an actual Unbounded Stream and more of an “On The Fly” Stream

References

- Stream API
  - http://download.java.net/jdk8/docs/api/java/util/stream/Stream.html
- Java 8 Explained: Applying Lambdas to Java Collections
  - http://zeroturnaround.com/rebellabs/java-8-explained-applying-lambdas-to-java-collections/
- Java 8 first steps with Lambdas and Streams
  - https://blog.codecentric.de/en/2013/10/java-8-first-steps-lambdas-streams/
- Java 8Tutorial: Lambda Expressions, Streams, and More
  - http://www.coreservlets.com/java-8-tutorial/

Questions?