Learning to Love the Lambda in the Stream

Introduction to Java 8 Lambda and Functional Interfaces

Speaker Introduction

- Richard Roda
- Sr. Technical Lead at DXC Technology
- Over 15 years of Java development experience
- OCA Java and Security+ certifications
- Linked In: https://www.linkedin.com/in/richardroda
- Twitter: @Richard_Roda
- These slides (pdf): https://tinyurl.com/love-lambda

What is a Lambda Expression?

- In Java, it is an unnamed function that may be bound to an interface as an object.
- Similar to a closure, class members, *effectively final* arguments and local variables are available to it.
- Lambdas may only exist when assigned to a Functional Interface, including being passed in as a parameter.
- An *effectively final* local variable or argument is either declared final, or is not changed such that if the final declaration were added, the code remains valid.

Lambda Examples

- Example 1a

```
Predicate<Integer> isFive = n -> n == 5;
System.out.println(isFive.test(4)); // false
```

- Example 1b

```
Predicate<Integer> mkTestFunc(int value)
    { return n -> n == value; }
Predicate<Integer> isFour = mkTestFunc(4);
System.out.println(isFour.test(4)); // true
```

- Lamdba expressions must be assigned to a functional interface
- `(n -> 5).test(); // Does not compile`

Functional Interface (FI) in Java 8

- "A functional interface is any interface that contains only one abstract method." — Oracle Java Tutorial
- The sole abstract method referred to as the *functional method*
- Example 2- Valid Functional Interface

```
@FunctionalInterface // Optional
public interface Example2 {
    int myMethod(); // Functional Method
    boolean equals(Object other); // In Object
    int hashCode(); // In Object
    default int myMethod2() {return myMethod();}
    static int myMethod3() {return 0;}
}
```

Binding Lambda to Example2 FI vs Anonymous Inner class

- Both of these implement myMethod defined in Example2.
- Since there is exactly one abstract functional method, method types and return values are inferred from the FI.

```
public class Example3 {
    static public void main(String[] args) {
        Example2 lambda = () -> 3; // 8 chars
        Example2 innerClass = new Example2() {
            @Override public int myMethod() {
                return 3;
            }
        }; // 5 lines of code, 65 chars
        System.out.println(lambda.myMethod()); // 3
        System.out.println(innerClass.myMethod()); // 3
    }
}
```

Functional Interface Conventions

- The abstract method is called the *functional method*
- The following conventions apply for type variables used by Java 8 FIs:
- T – First argument
- U – Second argument
- R – Return Value
- Any of the above are omitted if not used.
- If an FI lacks an argument, T is sometimes used for the return value instead of R.

Key Functional Interfaces

Predicate<T>

- Accepts an argument, returns a `boolean`.
- Commonly used to select matching elements, or filter for matching elements.
- Functional method: `boolean test(T t)`
- 2 argument FI: BiPredicate<T,U>
- Related Primitive FIs: DoublePredicate, IntPredicate, LongPredicate

Consumer<T>

- Accepts an argument. Returns no value (void).
- Commonly used to perform an operation, such as printing.
- Functional Method: `void accept(T t)`
- 2 Argument FI: BiConsumer<T,U>
- Related Primitive FIs: DoubleConsumer, IntConsumer, and Long Consumer

Supplier<R>

- Accepts no arguments, returns a result
- Commonly used to provide an origin value to an algorithm. Also a good interface to use for the Factory Object pattern.
- Functional Method: `R get()`
- Related Primitive FIs: DoubleSupplier, IntSupplier, LongSupplier

Function<T,R>

- Accepts an argument, returns a result.
- Commonly used to compute a result, or to map one value to another value.
- Functional Method: `R apply(T t)`
- 2 Argument FI: BiFunction
- Related Primitive FIs: [Double,Int,Long]Function, [Double,Int,Long]To[Double,Int,Long]Function, To[Double,Int,Long]Function, To[Double,Int,Long]BiFunction

UnaryOperator<T>

- Accepts an argument, returns the same type of result as its argument.
- Used to compute a result or map a value to the same type as the input.
- Functional Method: `T apply(T t)`
- 2 Argument FI: BinaryOperator<T>
- Related Primitive FIs: [Double,Int,Long]UnaryOperator, [Double,Int,Long]BinaryOperator
- UnaryOperator<T> extends Function<T,T> and BinaryOperator<T> extends BiFunction<T,T,T>

Comparator<T>

- Accepts two arguments, and returns an integer.
- Used to compare objects, and to impose a *total ordering* on a collection of objects.
- Functional Interface: `int compare(T o1, T o2)`
  - When o1 < o2, returns <= -1
  - When o1 = o2, returns 0
  - When o1 > o2, returns >= 1
- Even though Comparator has been around since the early days, it is a functional interface because it has a single abstract method.

Stream

Not to be confused with IO Streams

Java Stream Definition

- Abstraction for computation of elements. Is not a data structure, but rather a computation structure.
- A stream consists of
  1. A data source, such as a collection, file, or computation. May be infinite, such as the set of numbers starting at 0. A data source is *lazy*.
  2. Zero or more intermediate operations.
     - Accepts a stream and returns a another stream with the operation appended to it.
     - *Lazy*: Only executed when a terminal operation processed the stream.
  3. A terminal operation
     - Returns a result, such as a number or a collection.
     - *Eager:* It requests the elements from the final stream, which has the effect of pulling elements from the data source and applying the intermediate operations to them. A stream is a passive description of a computation until a terminal operation is applied.
     - Closes the stream. Any further operations are invalid and result in an IllegalStateException.

Add a collection of numbers

- Given `Collection<Integer> numbers` that has integers from 1 to 10, add the collection.
- For Loop

```
int total = 0;
for(Integer number : numbers) {total += number;}
return total; // 500500
```

- Stream reduction

```
return numbers.stream().reduce(0, (i,sum) -> i+sum); // 500500
```

- Same Sum using an IntStream

```
return IntStream.rangeClosed(1, 1000).sum(); // 500500
```

Breaking Down the Stream

```
stream(Collection<Integer> numbers) {
    return numbers.stream() // Data Source
        .reduce(0, (i,sum) -> i+sum); // Terminal Operation
}
```

- All streams have a data source, zero or more intermediate operations, and a terminal operation.
- `numbers` collection is the data source.
- `reduce` is a terminal *reduction* on the stream.
- A reduction distills all of the values in a given stream to a single value.
- Integer reduction examples: sum, average, median, min, and max.
- The first argument to `reduce` is the identity argument. For addition, it is 0. For a multiplication lambda it would be 1.
- The lambda is a `BinaryOperator<Integer>` that is given a running total and the current element. They are processed by adding them together.

Primitive Streams

- `IntStream`, `LongStream`, and `DoubleStream`
- Offers a performance benefit over generic stream by avoiding boxing of primitive computations.
- Offers additional methods, such as `sum()`, `min()`, `max()`, `average()`, and `summaryStatistics()`.
- Can replace a traditional for loop

```
IntStream.range(0, 10).forEach(System.out::println); // Print 0-9
```

- Use `mapToInt`, `mapToLong`, `mapToDouble`, and `mapToObj` to convert an existing stream to an `IntStream`, `LongStream`, `DoubleStream`, and `Stream<T>` respectively.
- The `mapToObj` can also convert `Stream<T>` to `Stream<R>` where `T` and `R` are different types.

New Requirement
Only Process Numbers divisible by 4

- Only process numbers divisible by 4.
- The `filter` intermediate operation creates a new stream with the contents of the previous stream where the `Predicate<T>` or primitive predicate is `true`.

```
int sum = IntStream.range(0, 1000) // Data Source
.filter(i -> i % 4 == 0) // Intermediate Operation
.sum(); // Reduction - Terminal Operation
System.out.println(sum); // 124500
```