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, *effectively final* arguments, local variables and class members are available to it.
- Lambdas may only exist when assigned to a Functional Interface, including being passed in as a parameter.

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
- Example 2- Valid Functional Interface

```
@FunctionalInterface // Optional
public interface Example2 {
    int myMethod(); // Abstract.
    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 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
        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:
- 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
- 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>

Streams

- 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 new 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:* Applying a terminal operation to a stream begins the process of pulling data from the data source, and applying the intermediate operations.
- Example:

```
System.out.print(IntStream.iterate(0, i -> i+1)
        .limit(12).sum()); // 66
```