Learning to Love the Lambda in the Stream

Introduction to Java 8 Lambda and Functional Interfaces

Speaker Introduction

- Richard Roda
- Sr. Developer at USANA Health Sciences
- Over 15 years of Java development experience
- Oracle Certified Professional Java 8
- 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 is 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 or returned as a result.
- 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 -> n == 5).test(4); // Does not compile`
- `var unknownType = n -> n == 5; // Does not compile`
- `var predicateType = mkTestFunc(4);// Compiles`

Lambda Syntax

- A lambda expression may take one of the following forms:
  - *[Argument List]* **->** *statement*
  - *[Argument List]* **->** { *statements*; **return;** }
- Argument List may take one of the following forms:
  - () -> …
  - i -> …
  - (i) -> …
  - (Integer i) -> …
  - (i,j…) -> …
  - (Integer i, String j…) -> …
- When using the { *statements*; **return;** } form, the return is optional for a void return value. When using a single *statement*, the result of the statement is implicitly returned.

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 is 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
    }
}
```

Key Functional Interfaces

Used by Streams

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 or the return value matches the argument(s), T is used for the return value instead of R.
- Many FIs that take one argument have a corresponding two argument version prefixed with “Bi”.

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 `LongConsumer`
- Collections and Streams have a forEach method to apply an action to each of their elements:

```
void forEach(Consumer<? super T> action)
```

- Can replace a for loop. Useful for implementing a form of the Visitor pattern.

Supplier<T>

- Accepts no arguments, returns a result
- Commonly used to provide an initial value to an algorithm, and as a source for multiple values.
- Functional Method: `T get()`
- Related Primitive FIs: `DoubleSupplier`, `IntSupplier`, `LongSupplier`
- Useful for implementing the Factory pattern.

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<T,U,R>`
- 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> & BinaryOperator<T>

- A specialization of function: 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>`
- `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 Method: `int compare(T lhs, T rhs)`
  - When lhs < rhs, returns < 0
  - When lhs = rhs, returns 0
  - When lhs > rhs, returns > 0
- Even though Comparator has been around since the early days, it is a functional interface that is used by the stream framework for sorting data.

Optional<T> Class

- Container class returned by various stream methods.
- Represents a value that may or may not exist. Used instead of returning a **null** value.
- `isPresent` returns **true** when a value exists, `ifPresent` executes a `Consumer<T>` when a value exists, `getOrElse` obtains the value or returns a specified value which may be **null** if does not exist, and `get()` obtains the value, throwing NoSuchElement if it does not exist.
- ```
  Optional<Integer> found = Stream.of(1,2).max((a,b) -> a - b);
  System.out.print("Found: " + found.isPresent() + ", value = ");
  found.ifPresent(System.out::println); // Found: true, value = 2
  ```
- ```
  Optional<Object> notFound = Stream.of().max((a,b) -> 0);
  System.out.print("Found: " + notFound.isPresent());
  notFound.ifPresent(v -> System.out.println(v)); // Found: false
  ```

Method Reference

Shorthand for lambdas that invoke a single method

Method Reference

- Shorthand for a Lambda that only calls a method
- Types of References
  - Static method, such as `String::valueOf`
  - Method on an instance, such as `System.out::println`
  - Constructor reference, such as `StringBuilder::new`
  - Instance method, such as `String::toUpperCase`
- Once familiar with syntax, these can often be read and understood faster.
- A method reference may always be transformed into a lambda, but a lambda may not always be transformed into a method reference.

Static Method Reference

- Example:

```
// public static valueOf(char[] data)
Function<char[],String> valueOf = String::valueOf;
// valueOf = s -> String.valueOf(s);
String value = valueOf.apply(new char[]
{'H','e','l','l','o'});
System.out.println(value); // Hello
```

- Arguments are bound in declaration order.

Method Reference on an Instance

- Example:

```
// public void print(Object x)
Consumer<Object> printer = System.out::print;
// printer = i -> System.out.print(i);
Arrays.asList(1,2,3,4).forEach(printer); // 1234
```

- The same rules that apply for binding lambda variables to a static method also apply when binding to a method on an instance:
- class members, *effectively final* arguments and local variables may be used as a method reference on an instance.
- Arguments are bound in declaration order.

Constructor Method Reference

- Example:

```
// public StringBuilder()
Supplier<StringBuilder> supplier = StringBuilder::new;
// supplier = () -> new StringBuilder();
StringBuilder sb = supplier.get().append("Hi!");
System.out.println(sb); // Hi!
```

- Creates a new instance of the class, and returns it as the result.
- Must be bound to a functional interface with a non-void return type.
- Supplier is canonically used for a constructor method reference.
- Function and BiFunction may be used for constructors with arguments.

Instance Method Reference

- Example:

```
// public String toUpperCase()
UnaryOperator<String> toUpper = String::toUpperCase;
// toUpper = s -> s.toUpperCase();
System.out.println(toUpper.apply("abc")); // ABC
```

- The first argument of the lambda becomes the instance the method reference operates on.
- The remaining arguments are bound in the order they occur.
- The first argument rule has significance when choosing the order of arguments for the “Bi” family of Functional Interfaces.

Streams

Java’s answer to Monads

Not to be confused with IO Streams

What is a Java Stream?

- Abstraction for computation of elements.
- A computation structure, not a data structure.
- A stream consists of
  1. A data source, such as a collection, file, stream builder, 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 processes the stream.
  3. A terminal operation
     - Often returns a result, such as a value 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.

Streams are Like Factory Conveyor Belts

- The data source is the raw material to be processed.
- Adding the intermediate operations is like getting the workers into place. The terminal operation is like the worker who packages the finished product.
- Like a conveyor belt takes the result of the previous worker's changes to the next worker, a Stream takes the data source output or previous intermediate operation result as the input to the next intermediate operation or terminal operation.
- A conveyor belt doesn't start until all the workers are in place and ready. Likewise a stream doesn't start until all the intermediate operations and the terminal operation have been defined.
- Defining the terminal operation starts the processing. Once it is running, it can't be changed.

Reduction – Add a Collection of Numbers

- A reduction is an operation that computes a value by processing all the values in the stream.
- Given `Collection<Integer> numbers` that has integers from 1 to 1000, add the collection.
- For Loop

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

- Stream reduction (using a `BinaryOperator<Integer>`)

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

- The lambda `(i,sum) -> i+sum` is a *pure function* because it only reads its arguments, does not change any external state (no side-effects), and always returns the same value for the same arguments. For example: `apply(3,4)` always returns 7.
- Pure functions are inherently thread safe and should be used with streams whenever possible. Otherwise, nondeterministic and unpredictable behavior may occur. Consumers and Suppliers are notable exceptions to this rule.

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 processes 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 property. For addition and counting, it is 0. For a multiplication it is 1, for strings it is “” (empty string).
- 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`
- They offer a performance benefit over the generic stream by avoiding boxing of primitive computations.
- They offer additional terminal operations, such as `sum()`, `min()`, `max()`, `average()`, and `summaryStatistics()`.
- Can replace a traditional for loop with range and forEach.

```
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.
- Use the `boxed()` method to convert a primitive stream to its equivalent object stream by boxing the primitive values as follows:
  - `IntStream` to `Stream<Integer>`
  - `LongStream` to `Stream<Long>`
  - `DoubleStream` to `Stream<Double>`

Intermediate Operations

These Create a New Stream with the Operation Appended to It

Map

- Not to be confused with `java.util.Map`.
- Uses a `Function<T,R>` or related Primitive FIs to apply a computation or mapping on stream elements.
- A pure function should be used if possible.
- May change the element type of a stream by returning values of a different type.

```
Stream<Character> s = IntStream.range(65, 75)
    .mapToObj(i->(char)i); // Stream<Character>
s.forEach(System.out::print); // ABCDEFGHIJ
```

- Change values, but keep data type (int).

```
IntStream.range(0, 10).map(i -> i*10)
.forEach(System.out::println); // 0, 10 ... 90
```

Filter

The `filter` intermediate operation creates a new stream with the contents of the previous stream where the `Predicate<T>` or primitive predicate is `true`.

```
IntSummaryStatistics summaryStatistics =
IntStream.range(0, 1000) // Data Source
.filter(i -> i%4 == 0) // Intermediate Operation
.summaryStatistics(); // Terminal Operation
System.out.println(summaryStatistics);
/* count=250, sum=124500, min=0,
average=498.000000, max=996 */
```

Limit and Skip – Infinite Streams

- Limit intermediate operation limits the values produced by a stream. An infinite stream becomes a finite stream.
- Skip intermediate operation skips the specified elements
- Order of these operations matters
  - Skip before limit – Skipped items not counted against limit
  - Skip after limit – Skipped items counted against limit
- ```
  IntStream.iterate(0, i -> i+1).skip(4).limit(6)
  .forEach(System.out::print); // 456789
  ```
- ```
  IntStream.iterate(0, i -> i+1).limit(6).skip(4)
  .forEach(System.out::print); // 45
  ```
- `IntStream.iterate` uses an initial value with an `IntUnaryOperator` to create an infinite stream.

Terminal Operations

Let's Get This Party Started. Let’s Get This Stream Processing

Terminal Operations

- count - A *reduction* that returns the number of elements in the stream. Never use on an infinite stream.
- reduce - Perform a *reduction* of the stream using a `BinaryOperator` to accumulate the elements. Never use on an infinite stream.
- anyMatch - Returns **`true`** and stops processing if any element matches the supplied `Predicate`, **`false`** otherwise. Empty Stream is **`false`**.
- allMatch - Returns **`false`** and stops processing if any element does **not** match the supplied `Predicate`, **`true`** otherwise. Empty Stream is **`true`**
- noneMatch - Returns **`false`** and stops processing if any element matches the supplied `Predicate`. Empty Stream is **`true`**.
- forEach - A `void` operation that presents each element to a `Consumer` for processing.
- A reduction is an operation that computes a single value by processing all the values on the stream. Never reduce an infinite stream.

Terminal Operations that return Optional<T>

- These terminal operations return an `Optional<T>` because the value does not exist in an empty stream.
- findFirst - produces the first element in a stream. Because this implies ordering of the elements, any parallel stream is transformed into a sequential stream to guarantee element encounter order
- findAny - produces any element on the stream. It does not impose any overhead on parallel stream, but may produce differing values on invocation of the same stream.
- min - produces the minimum element
- max - produces the maximum element.

Collector (Terminal Operation)

A Mutable Reduction That Creates an Object to Process All Stream Elements

Never Use on an Infinite Stream

Collections Collectors

- These collectors take the elements and add them to a collection.
- There are toList(), toSet(), and toCollection() collectors.
- ```
  List<Integer> ints = IntStream.of(1,2,2,3,4,5).boxed()
  .collect(Collectors.toList()); System.out.println(ints);
  // [1, 2, 2, 3, 4, 5]
  ```
- ```
  Set<Integer> intSet = IntStream.of(1,2,2,3,4,5).boxed()
  .collect(Collectors.toSet()); System.out.println(intSet);
  // [1, 2, 3, 4, 5]
  ```
- ```
  // Custom collection type with a sort applied to it.
  LinkedHashSet<Integer> sortedSet = IntStream.of(1,2,2,3,4,5)
  .boxed().sorted(Comparator.reverseOrder())
  .collect(Collectors.toCollection(LinkedHashSet::new));
  System.out.println(sortedSet);
  // [5, 4, 3, 2, 1]
  ```
  ```

Partition Collector

- The Partition collector uses a `Predicate<T>` to create a map with the keys **`false`** and **`true`**.
- Both the **`false`** and **`true`** key and value always exist in the map even if the corresponding value is not present. In such a case, the value will be an empty collection, an empty optional, or a sum or count of 0.
- Use the predicate in the previous example to create a map with elements divisible by 4 and not divisible by 4.

```
Map<Boolean,Integer> summap =
IntStream.range(0, 1000).boxed()
.collect(partitioningBy(i -> i%4==0, summingInt(i -> i)));
System.out.println(summap); // {false=375000, true=124500}
```

The `summingInt` collector is an example of a *downstream collector*. In this case, it accepts the values of the partitioning by collector and produces a sum reduction of the values.

Grouping By Collector

- For the next example, consider the following stream producing function
- `static Stream<String> aboutJack() {`

```
return Stream.of("All","work","and","no","play","makes",
"jack","a","dull","boy","but","all","play","and","no",
"work" ,"makes","jack","a","fool"); }
```

- Group each word by starting letter, in alphabetical order

```
aboutJack().sorted().collect(
    Collectors.groupingBy(s -> s.charAt(0),
    TreeMap::new, Collectors.toCollection(TreeSet::new)));
/* A=[All], a=[a, all, and], b=[boy, but], d=[dull],
f=[fool], j=[jack], m=[makes], n=[no], p=[play], w=[work] */
```

Grouping By Concurrent

- Streams may be processed in parallel by using the `parallel` method using concurrent collectors and data structures.

```
aboutJack().parallel().collect( Collectors.groupingByConcurrent(
s -> s.charAt(0), ConcurrentSkipListMap::new,
Collectors.toCollection(ConcurrentSkipListSet::new)));
/* A=[All], a=[a, all, and], b=[boy, but], d=[dull], f=[fool],
j=[jack], m=[makes], n=[no], p=[play], w=[work] */
```

- Count the words
- `aboutJack().parallel().collect(Collectors.groupingByConcurrent(`

```
s -> s, ConcurrentSkipListMap::new, Collectors.counting()));
/* All=1, a=2, all=1, and=2, boy=1, but=1, dull=1, fool=1, jack=2,
makes=2, no=2, play=2, work=2 */
```

Joining Collector

- A process where a stream of `CharSequence` is concatenated together to form a string. Recall the `aboutJack` stream:

```
static Stream<String> aboutJack() { return Stream.of(
"All","work","and","no","play","makes","jack","a","dull",
"boy","but","all","play","and","no","work","makes",
"jack","a","fool"); }
```

- Join this into words separated with a space:

```
aboutJack().collect(Collectors.joining(" "));
/* All work and no play makes jack a dull boy but all play
and no work makes jack a fool */
```

AutoClosable Lambdas

Use try-with-resources with any class, and catch the close exception

AutoClosable is a Functional Interface

- `public interface AutoCloseable {`
- `    void close() throws Exception;`
- `}`
- This interface is a functional interface (FI) because it has exactly one abstract method.
- The Functional Method is: `void close()`.
- The missing `@FunctionalInterface` annotation is unnecessary.

Use try-with-resources with any class Example: Close a Context

- In Java 7, try-with-resources was added to the language.
- Unfortunately, not every class that could benefit from it implemented it.
- Using Lambdas, anything can leverage try-with-resources.

```
public void useContext(Context ctx) throws Exception {
    try(AutoCloseable it = ctx::close) {
        doSomethingWithContext(ctx);
    }
}
```

Issues with the AutoClosable Functional Interface (FI)

- The close method throws Exception.
- The thrown Exception will either need to be caught or processed.
- This may result in the code being littered with unnecessary catch statements.

Fixing the AutoClosable FI

- If we wrote our own Closable interface:

```
public interface NamingClosable extends AutoCloseable {
    @Override public void close() throws NamingException;
}
```

- Then we can write

```
public void useContext(Context ctx) throws NamingException
{
    try(NamingClosable it = ctx::close) {
        doSomethingWithContext(ctx);
    }
}
```

Parameterizing AutoClosable Exceptions

- Using generics, it is possible to parameterize the checked exceptions that a sub-interface of AutoClosable may throw.
- This example demonstrates how to parameterize a single checked exception.

```
public interface CloseIt1<E extends Exception>
extends AutoCloseable {
    default void close() throws E { closeIt(); }
    void closeIt() throws E;
}
```

- The default `close()` method is necessary because applying the generic to an abstract `close()` method results in a compiler error when used in a try-with-resources statement.

Using the Parameterized FI

- Using `CloseIt1` from the previous slide:

```
public void useContext(Context ctx) throws NamingException {
    try(CloseIt1<NamingException> it = ctx::close) {
        doSomethingWithContext(ctx);
    }
}
```

- The close method of the Context is bound to the CloseIt1 resource. The try-with-resources feature of Java does the heavy lifting of the resource exception processing.

Decorator Pattern

- One of the core patterns introduced in the *Design Patterns, Elements of Reusable Object Oriented Software* by Gamma, Helm, Johnson, and Vlissides.
- Pattern allows behavior to be added to an object dynamically, by decorating it, or wrapping it with another object of the same abstract type (such as an interface).
- This pattern may be leveraged to add capabilities to AutoClosables, such as exception handling.
- Since AutoClosable is a Functional Interface, the decorator may be expressed as a lambda.
- https://en.wikipedia.org/wiki/Decorator_pattern

Decorating the Close Lambda

- Consider the following code
  - Assume `NotClosedException` is an unchecked exception with an accessible constructor that takes a Throwable.

```
public interface CloseIt0 extends AutoCloseable {
    public void close() throws NotClosedException;
    public static CloseIt0 wrapAllException(AutoCloseable
    autoCloseable) {
        // Decorating with a lambda that wraps all Exceptions
        return () -> { try { autoCloseable.close(); }
        catch (Exception ex) { throw new NotClosedException(ex);}
        };
     }
    }
```

Catching the Decorated Close Exception

- This close lambda is decorated to wrap any exceptions that occur within a `NotClosedException`. If no exception occurs within the body, this wrapped exception will be caught and processed by the catch clause. Otherwise, it will be a suppressed exception.

```
public void useContext(Context ctx) throws NamingException {
    try(CloseIt0 it = CloseIt0.wrapAllException(ctx::close)) {
        doSomethingWithContext(ctx);
    } catch (NotClosedException ex) {
        logger.log(Level.WARNING, ex.getCause().getMessage()
        , ex.getCause());
    }
}
```

The CloseIt Project

- Provides generic functional interfaces extending `AutoCloseable` to use as the target of try-with-resources lambdas. Supports 0-5 checked exceptions.
- Makes it easy to use try-with-resources for any object that needs cleanup. May replace the try-finally construct.
- Provides these decorators for handling close exceptions
  - Ignore – Pretend the exception never happened. Discard it.
  - Consume – Do something, such as log the exception, then discard.
  - Rethrow – Do something, such a log the exception, then throw it.
  - Hide – Hide a checked exception from the compiler and throw it.
  - Wrap – Wrap the exception within another exception of a different type. This a form of the Adapter design pattern. https://en.wikipedia.org/wiki/Adapter_pattern.

Questions

- Oracle’s Lambda Quick Start Tutorial: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- These slides (pdf): https://tinyurl.com/love-lambda
- CloseIt: https://github.com/RichardRoda/closeit - com.github.richardroda.util:closeit:1.6
- This Project: https://github.com/RichardRoda/2017-CodePaLOUsa-Lambda
- My Linked In: https://www.linkedin.com/in/richardroda
- My Twitter: @Richard_Roda
- These slides license: CC BY 3.0 US license terms