Contents

Java lambdas and key functional interfaces in the standard library

Introduction

Java 8 added a handful of key features to the Java language, accompanied by additions to the standard library. Among the language features added were *lambdas*, often called *anonymous functions* or *closures*. In order to implement this feature in a manner consistent with the historical aims and use of the language, it was decided that lambdas would be restricted to implementation of *functional interfaces*—interfaces with exactly one unimplemented method. Such interfaces had already existed in the standard library, but the distinction was formalized in Java 8, and several new functional interfaces were added to the standard library. This document is an overview of some of the functional interfaces, with lambda-based implementation examples.

Note: Due to the lambda-centric focus of this document, *Comparable<T>*, *Iterable<T>*, *Closeable*, and *AutoCloseable* are not addressed, even though they are formally functional interfaces (since each has exactly one unimplemented method). The purposes of those interfaces essentially dictate that they perform computations based on the *states* of instances of the implementing classes; thus, they are not well-suited to implementation via lambdas, which don't define accessible state.

Runnable (java.lang)

- Primarily intended for use as a unit of processing operations that can be executed in a separate thread from the originator. For example, an instance of a *Runnable* implementation may be passed to a new `Thread` in the `Thread(Runnable)` constructor; when the `Thread.start()` method is invoked, the new execution thread is started, and the *Runnable.run()* method is invoked in that thread.
- The method to be implemented is `void run()`. Thus, an implementing lambda must take no parameters, return no value, and throw no checked exceptions.
- While *Runnable* is intended for use as a basic "atom" of concurrent processing, it is not required to be used in a concurrent mode. In particular, unlike `Thread.start()`, *Runnable.run()* may be invoked multiple times, from any thread.

Example: Create *Runnable* and invoke `run` method on threads.

```
Runnable identify = () ->
    System.out.printf("%d: %s%n", System.currentTimeMillis(),
        Thread.currentThread().getName());
Thread t = new Thread(identify);
identify.run();
t.start();
```

In this example, a reference to the lambda implementation of *Runnable* is declared and initialized as the local variable `identify`. A `Thread` is created, passing this lambda as a constructor parameter. Then `identify.run()` is invoked on the main thread, after which the new thread is started, causing `identify.run()` to be invoked on that thread.

The output from the above will be slightly different every time, but it will generally be something like

```
1587678772815: main
1587678772840: Thread-6
```

Since both threads (`main` and—in the example output above—`Thread-6`) are writing to a shared resource (the standard output device accessed via `System.out`), it's actually possible to get interleaved output between the 2 threads.

Predicate<T> (java.util.function)

- Typically used for filter operations on a *Collection* or *Stream*.
- Parameterized by the type contained in the *Collection* or *Stream*.
- Method to be implemented is `boolean test(T t)`. Thus, a lambda must take a single parameter and return/evaluate to a `boolean` result.

Examples: Filter `Stream` or `Collection`...

... using *Collection*.removeIf(*Predicate*).

```
Collection<Integer> data =
    new LinkedList<>(List.of(1, 1, 2, 3, 5, 8, 13));
data.removeIf((v) -> v % 2 == 0); // Remove even values.
// data contains [1, 1, 3, 5, 13]
```

... using *Stream*.filter(*Predicate*).

```
List<Integer> data = Stream.of(1, 1, 2, 3, 5, 8, 13)
    .filter((v) -> v % 2 == 1) // Preserve odd values.
    .collect(Collectors.toList());
// data contains [1, 1, 3, 5, 13]
```

Note that the sense of the predicate is reversed in the two uses shown above: *Collection.removeIf(Predicate)* **removes** an item from the collection if (and only if) the predicate evaluates to `true` for that item; *Stream.filter(Predicate)* **preserves** an item in the stream if (and only if) the predicate evaluates to `true` for that item.

Consumer<T> (java.util.function)

- Typically used to perform a common operation on all items in a *Collection* or *Stream*, or on the value which may be contained in an `Optional`. (It can be useful to view *Collection.forEach(Consumer)* as an "inside-out" variation on the *enhanced-for*.)
- Parameterized by the type contained in the *Collection*, *Stream*, `Optional`, etc.

- Method to be implemented is `void accept(T t)`. Thus, a lambda must take a single parameter and return no result. (If an expression lambda is used, the evaluated result will be ignored.)

Examples: Print all items in *`Collection`* or *`Stream`*...

... using *`Collection`*`.forEach(`*`Consumer`*`)`.

```
List<Integer> data =
    new LinkedList<>(List.of(1, 1, 2, 3, 5, 8, 13));
data.forEach(System.out::println); // Print each value.
```

... using *`Stream`*`.forEach(`*`Consumer`*`)`.

```
Stream.of(1, 1, 2, 3, 5, 8, 13)
    .forEach(System.out::println); // Print each value.
```

Both of these examples produce this output:

```
1
1
2
3
5
8
13
```

`Supplier<T>` (`java.util.function`)

- Used to provide a source of values for a *`Stream`*, alternative values for an `Optional`, etc.
- Method to be implemented is `T get()`. Thus, a lambda must take no parameters and return/evaluate to the parameterized type `T`.
- Implemented by a lambda in some cases; however, since state may often need to be maintained across invocations of `get()`, *`Supplier`* is sometimes implemented via a class with the relevant state fields.
- When using *`Stream.generate(Supplier)`*, the size of the resulting stream must generally be controlled through *`Stream.limit(long)`* or another operation that truncates the stream in some fashion.

Example: Generate *Stream* of random numbers using *Stream*.generate(*Supplier*).

```
Stream.generate(Math::random) // Get each value in turn.
    .limit(100) // Take only the first 100 values.
    .forEach(System.out::println); // Print each value.
```

This example uses a *Supplier<Double>*, implemented as a lambda (expressed in method reference syntax), to generate a *Stream<Double>* of random values. The stream is truncated after 100 items, and then each item is printed via a *Consumer<Double>* lambda, expressed with method reference syntax.

Function<T,R> (java.util.function)

- Used to map a value of one type to a value of another, or to mutate an object in-place (returning the mutated object rather than a new object).
- Often employed to transform all of the items in a *Stream*, using *Stream.map(Function)*.
- Parameterized by the type of the original value (*T*) and the type of the mapped value (*R*); these may, in fact, be the same type. (*UnaryOperator<T>* extends *Function<T,R>* explicitly for this purpose.) If used in *Stream.map(Function)*, then this also has the effect of changing a *Stream<T>* to a *Stream<R>*.
- Method to be implemented is `R apply(T t)`. Thus, a lambda implementation must take a single parameter of type `T`, and return a value of type `R`. …

Example: Map *Stream* of random values to dice rolls using *Stream*.map(*Function*).

```
Stream.generate(() -> Math.random())
    .limit(100) // Take only the first 100 values.
    .map((v) -> 1 + (int) (6 * v)) // Map from [0, 1) to {1 … 6}.
    .forEach(System.out::println); // Print each value.
```

The example uses a lambda implementing *Supplier<Double>* to generate a *Stream<Double>* of pseudo-random numbers. Then, a lambda implementation of *Stream.map(Function)* is used to map the `Double` values to `Integer` values in the range from 1 to 6 (inclusive)—i.e. random dice roll values. (Note that this example includes 2 auto-boxing operations, and 1 auto-unboxing operation. Can you spot them?)

BinaryOperator<T> (java.util.function)

- Among other uses, employed in multiple overloads of *Stream.reduce* to combine pairs of values in the stream, ultimately resulting in producing a single value.
- The method to be implemented is `T apply(T t1, T t2)`; a lambda implementation must therefore take 2 parameters of type `T` and return a value of type `T`. (This is a subinterface of *BiFunction<T,U,R>*; the `apply` method of that interface takes 2 parameters, of types `T` and `U`, respectively, and returns a result of type `R`.)

Example: Sluggify `String` using *Stream*.reduce(*BinaryOperator*).

```
Pattern delimiter = Pattern.compile("\\W+");
String source = "The quick brown fox jumps over the lazy dog.";
delimiter.splitAsStream(source) // Split on non-word characters.
    .filter((s) -> !s.isEmpty()) // Filter out empty strings.
    .map(String::toLowerCase) // Map each string to lower-case.
    .reduce((s1, s2) -> s1 + "-" + s2) // Concatenate into "slug".
    .ifPresent(System.out::println); // Print the value.
```

This example splits a `String` into a *Stream<String>* using a regular expression `Pattern`. Then, empty items are filtered out (using a *Predicate<String>* lambda) and the remaining values are converted to lower-case (using a *Function<String, String>* lambda). Finally, the *Stream<String>* contents are reduced to an `Optional<String>` using a `BinaryOperator<String>` lambda, and the contents of the `Optional<String>` are printed using a *Consumer<String>* lambda (expressed with method reference syntax). This produces the output

```
the-quick-brown-fox-jumps-over-the-lazy-dog
```

(Note: A joining collector could have produced the same result as the reduce operation in the above example; see the *Stream.sorted(Comparator)* example, below, for an illustration.)

Comparator<T> (java.util)

- Used for ordering (sorting or maintaining the sorted order of) items in an array, *Collection*, or *Stream*.
- A *Comparator<T>* is typically used when a natural ordering is not defined (i.e. when `T` does not implement *Comparable<T>*), or as an alternative to natural order.

- An implementation can (and generally should) take advantage of comparison capabilities defined for the types that make up the state of `T`, as well as the static factory methods provided by *`Comparator`*.
- The method to be implemented is `int compare(T t1, T t2)`. A lambda implementation must therefore take 2 parameters of type `T`, and return an `int`, where a negative value implies that `t1 < t2` (for the purpose of ordering), a positive value implies `t1 > t2`, and zero means that `t1` and `t2` are equivalent for ordering purposes.

Examples: Sort array, *`Collection`*, or *`Stream`* by reciprocals ...

... using `Arrays.sort(T[],`*`Comparator`*`).

```
Integer[] data = {-1, 3, 2, 10, -5, 6};
Arrays.sort(data, (d1, d2) -> Double.compare(1.0 / d1, 1.0 / d2));
System.out.println(Arrays.toString(data));
```

... using *`List`*`.sort(`*`Comparator`*`).

```
List<Integer> data =
    new LinkedList<>(List.of(-1, 3, 2, 10, -5, 6));
data.sort((d1, d2) -> Double.compare(1.0 / d1, 1.0 / d2));
System.out.println(data);
```

... using *`Stream`*`.sorted(`*`Comparator`*`).

```
System.out.println(Stream.of(-1, 3, 2, 10, -5, 6)
    .sorted((d1, d2) -> Double.compare(1.0 / d1, 1.0 / d2))
    .map(String::valueOf)
    .collect(Collectors.joining(", ", "[", "]")));
```

All of the above produce the same output:

```
[-1, -5, 10, 6, 3, 2]
```