Java SE 8

Lambdas and Streams

Hands On Lab

Student Guide

Lambdas Lab: Student Guide

Introduction

Welcome to the Lambdas and Streams Java SE 8 Hands on Lab!

The idea of this lab is to provide you with a set of exercises, starting simply and gradually increasing in complexity that will demonstrate some of the many capabilities of the new Lambda syntax and Streams API that make up some of the biggest changes to Java SE 8 over previous versions.

Lambdas and Streams offer a very powerful and compact way of coding Java for many situations where you want to do things like search, filter or sort collections of data using a functional style of programming. Aside from the simplicity of syntax, one of the key elements of Lambdas and Streams is to remove the distinction in the way you code from either being explicitly serial or explicitly parallel. The libraries have been designed to make taking advantage of multi-core, multi-processor machines very simple.

Don't worry if you're not familiar with the concepts of functional programming or Lambda calculus, the exercises will show you practically how to use these great new features.

Getting Started

The following three sections of the lab guide provide some background material to help you understand how to use Lambda expressions and the Streams API. This information should be used in conjunction with the API documentation to give you the details you need to start solving the exercises.

If you'd like to dive straight in to the code and get your hands dirty skip to the section on *Running The Exercises* and work from there.

Lambda Expressions

Lambda expressions are the big new language feature in Java SE 8. The syntax is simple and once you get used to using them they are very straightforward.

A Lambda expression represents an anonymous function. This is very much like a method in Java that we are all familiar with but, unlike a method, it is not associated with a class. The structure of a Lambda expression is the same, in principal, as a method, in that it has a set of typed parameters, a body, a return type and potentially a set of thrown exceptions. The basic syntax is as follows:

```
(<optional-parameters>) -> lambda-body
```

Unlike methods Lambda expressions make use of type inference, so it is not necessary to have an explicit return statement (although you can use one if you want) and the parameters do not need to have the type stated explicitly if the compiler is able to infer the types from the use of the lambda expression.

Since a Lambda expression represents an anonymous function, what is its type? This is very important, since Java is a statically typed, object-oriented language.

The answer is that it depends on where the Lambda expression is used. Let's look at a couple of examples of a Lambda expression being used in an assignment and as a method parameter:

```
Predicate<Double> p = t -> t != currentTemperature;
```

and

```
filter(t -> t != currentTemperature)
```

In both examples the type of the Lambda expression is a Predicate<Double>. If you look at the API documentation you will find that Predicate is an *interface* that has only one abstract method (also referred to as a Functional Interface). You can't instantiate an interface in Java, so the Lambda expression's type is that of the interface and it provides an implementation of the single abstract method, but there is no actual object involved.

You can use Lambda expressions wherever the type required is a functional interface. Java has long had support for anonymous inner classes, which also provide a way to implement a single abstract method type, but with lots of boilerplate code and the automatic generation of an object. One key point that applies to both anonymous inner classes and Lambda expressions is how you can refer to variables from the surrounding scope. In the above examples we use the currentTemperature variable. For an anonymous inner class this would have to be marked final. For a Lambda expression it does not need to be explicitly marked as final, but must behave in the same way as if it was (i.e. it can only have its value assigned once). Variables from the surrounding scope used in Lambda expressions are said to be 'effectively final'.

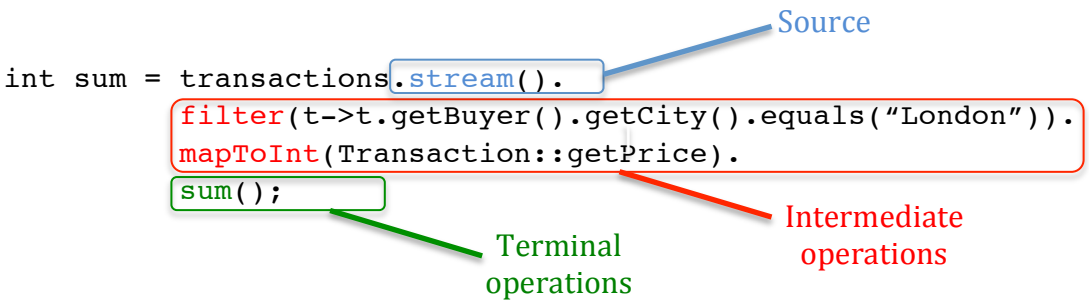
Note also that the parameter, t, does not have a type defined for it; this is because the compiler can infer this from the fact that the Predicate has a type parameter of Double (auto-boxing and unboxing take care of the object to primitive mapping that is required). Also as a single parameter the brackets around t are optional. If the abstract method takes no parameters an empty set of brackets must be used, e.g. `() -> new MyClass()`

Streams Basics

A Stream is a way of representing the operations you want to perform on a set of data. Where this data comes from does not matter and it is possible to have both finite and infinite streams.

You can think of a Stream as a pipeline that needs to be created to process your data in the desired manner. There are three key parts to a Stream:

1. A **source**: This is where the set of elements for the data come from. There are various ways of creating a Stream source as will be shown in the next section. The output of a Stream source is, not surprisingly, a Stream of elements from the source. This is the start of the processing pipeline.
2. Zero or more **intermediate operations**: Intermediate operations take a Stream as input and generate a Stream as output. This means that the output of one intermediate operation can be used as the input to another intermediate operation, thus linking them together to perform more complex processing.
3. A **terminal** operation: A terminal operation takes as input a `Stream`, but does not produce one as output. The output could be a concrete value, like a collection or integer, or could be a side effect like printing a particular message. This is the end of the processing pipeline. Some terminal operations in the Streams API (like `min()` and `max()`) return an `Optional`. This is an object that holds a reference to an object. The `Optional` class handles the situation where the reference is null without generating a `NullPointerException`. Please read the API documentation for more details.



Useful Stream APIs

Now you understand the basics of Lambda expressions and the Streams APIs let's look at some of the useful APIs you'll need for the exercises.

Stream Sources

- **From a Collection**. The Collection interface has been extended to include `stream()` and `parallelStream()` methods. The difference (as is probably obvious) is that `stream()` work serially and `parallelStream()` will decompose the Stream (where possible) into parallel operations and use the fork-join framework to execute them.
- **From an array**: An array is a collection of data; to create a Stream from an array use the `Arrays.stream()` or `Stream.of()` methods. Note there is no `parallelStream()` method, but a serial Stream can be converted to a parallel Stream by calling `parallel()` on the Stream.
- **From a File**: The BufferedReader class has been extended to include a new method, `lines()`. This will return a Stream that consists of the lines read by the BufferedReader, which can easily be instantiated to read from a file.
- **Explicitly, from a set of values**: The `Stream.of()` static method will create a Stream from the values that are passed as arguments to the method. This can be used with the `flatMap()` method (described below).
- A `Stream` of consecutive numeric values can be generated using the `IntStream` class (there are also complementary `DoubleStream` and `LongStream` classes). Unlike a `for` loop an `IntStream` can be converted to a parallel stream using `parallel()`.

java.util.function Package

The `java.util.function` package contains interfaces that are typically used as parameters to the methods of the Stream class. These will be the ones you will need to write Lambda expressions for to represent the single abstract method.

- **`Consumer`**: The abstract method, `accept()`, accepts a single argument and returns no result.

```
void accept(T t)
```

An example Lambda expression for this is

```
v  ->  System.out.println("Value = " + v)
```

- **`Function`**: The abstract method, `apply()`, accepts a single argument and produces a result. The type of the argument and result do not need to be the same.

```
R  apply(T t)
```

An example Lambda expression for this is:

```
t -> ((t – 32) * 5 / 9)
```

- **`Predicate`**: The abstract method, `test()`, accepts one argument and returns a boolean value.

```
Boolean test(T t)
```

An example Lambda expression for this is:

```
a -> a >= retirementAge
```

- **`Supplier`**: The abstract method, `get()`, takes no arguments and returns a result.

```
T  get()
```

An example Lambda expression for this is:

```
() -> new MyClass()
```

- **`BinaryOperator`:** Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
An example Lambda expression is:

```
(a, b) -> a + b
```

If you look at the API documentation you will find there are several variants of these interfaces that are type specific, e.g. `DoubleFunction` and `IntConsumer`.

There is one other interface you need to know about, which is `Collector`, which is part of the `java.util.stream` package. This is an interface (but not a functional one) that represents a mutable reduction operation that accumulates input elements into a mutable result container. Although you can write your own implementation most common uses can be handled by one of the methods available in the `Collectors` utility class, e.g. `toList()`, `toSet()` and `joining()`.

The Stream Class

The `Stream` class contains several intermediate and terminal operation methods.

Intermediate Methods

- **`distinct()`**: Returns a stream consisting of the distinct elements of the input stream.
- **`filter()`**: Returns a stream consisting of the elements of the input stream that match the given predicate.
- **`limit()`**: Returns a stream consisting of the elements of the input stream, truncated to the length specified by the parameter.
- **`skip()`**:Returns a stream consisting of the remaining elements of the input stream after discarding the number of elements specified by the parameter.
- **`sorted()`**: Returns a stream consisting of the elements of the input stream, sorted according to the provided `Comparator`. There is also a no parameter version of this that will sort based on natural order.
- **`map()`:** Returns a stream consisting of the results of applying the given function to the elements of the input stream. This is a one-to-one mapping function.
- **`flatMap()`:** Returns a stream consisting of the results of replacing each element of the input stream with the contents of a new stream produced by applying the provided mapping function to each element. Essentially this is a one-to-many mapping function, where each element on the input `Stream` creates a new `Stream` of results. Rather than returning a stream-of-streams, `flatMap` 'flattens', or concatenates all the Streams created into one output `Stream`.
- **`peek()`:** Returns the same output `Stream` as the input `Stream`, but applies the `Consumer` specified by the parameter effectively as the elements go past.

Terminal Methods

- **`anyMatch()`, `allMatch()`:** Returns whether any/all elements of the input `Stream` match the provided `Predicate`.
- **`findFirst()`:** Returns an `Optional` containing the first element of this `Stream`, or an empty `Optional` if the stream is empty.
- **`findAny()`:** Like `findFirst()`, but inherently non-deterministic. If the input `Stream` is parallel the first element from any thread that returns a result. If used multiple times on the same data set the result will not therefore be guaranteed to always be the same.
- **`noneMatch()`:** Returns whether no elements of the input `Stream` match the provided `Predicate`.
- **`collect()`:** Performs a mutable reduction operation on the elements of the input stream using a `Collector`.
- **`count()`:** Returns the count of elements in the input `Stream`.

- **`forEach()`:** Performs the action specified by the `Consumer` parameter for each element of the input `Stream`.
- **`min()`, `max()`:** Returns the minimum/maximum element of the input `Stream` according to the provided `Comparator` in the form of an `Optional`. There are no-arg versions that use natural ordering.
- **`reduce()`:** Performs a reduction on the elements of this `Stream` and returns an `Optional` containing the reduced value, if any. There are three variants of this method. Read the API documentation for more detail.
- **`toArray()`:** Returns an array containing the elements of this `Stream`.

Collectors

The `Collectors` utility class provides a range of useful methods that eliminates a lot of the need to write your own `Collector`, which requires you to specify a `supplier()`, `accumulator()`, `combiner()` and `finisher()`.

- **`toList()`**: Returns a Collector that accumulates the input elements into a new List.
- **`toMap(Function keyMapper, Function valueMapper)`**: Returns a `Collector` that accumulates elements into a `Map` whose keys and values are the result of applying the provided mapping functions to the input elements.
- **`groupingBy(Function classifier)`**: Returns a `Collector` implementing a "group by" operation on input elements , grouping elements according to a classification function, and returning the results in a `Map`.
- **`groupingBy(Function classifier, Collector downStream)`**: Returns a `Collector` implementing a cascaded "group by" operation on input elements, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream `Collector`.
- **`counting()`**: Returns a `Collector` that counts the number of input elements.
- **`joining()`**: Returns a `Collector` that concatenates the input elements into a `String`, in encounter order.
- **`joining(CharSequence delimiter)`**: Returns a `Collector` that concatenates the input elements, separated by the specified delimiter, in encounter order. NOTE: You can conveniently use a `String` to specify the `CharSequence`.

Comparator

`Comparator` is a functional interface, but in Java SE 8 with the introduction of default methods and the inclusion of static methods in interfaces has multiple methods. Several of these are very useful for the exercises:

- **`naturalOrder()`**: Returns a comparator that compares Comparable objects in natural order.
- **`reverseOrder()`**: Returns a comparator that imposes the reverse of the natural ordering.
- **`comparing(Function keyExtractor)`**: Accepts a function that extracts a `Comparable` sort key, and returns a `Comparator` that compares by that sort key. There are type specific versions of this method, `ComparingInt`, `comparingDouble` and `comparingLong`.
- **`thenComparing(Comparator other)`**: Returns a lexicographic-order comparator with another comparator. This can be used to combine Comparators to sort by, for example alphabetic order and then length order.

Useful Collections Methods

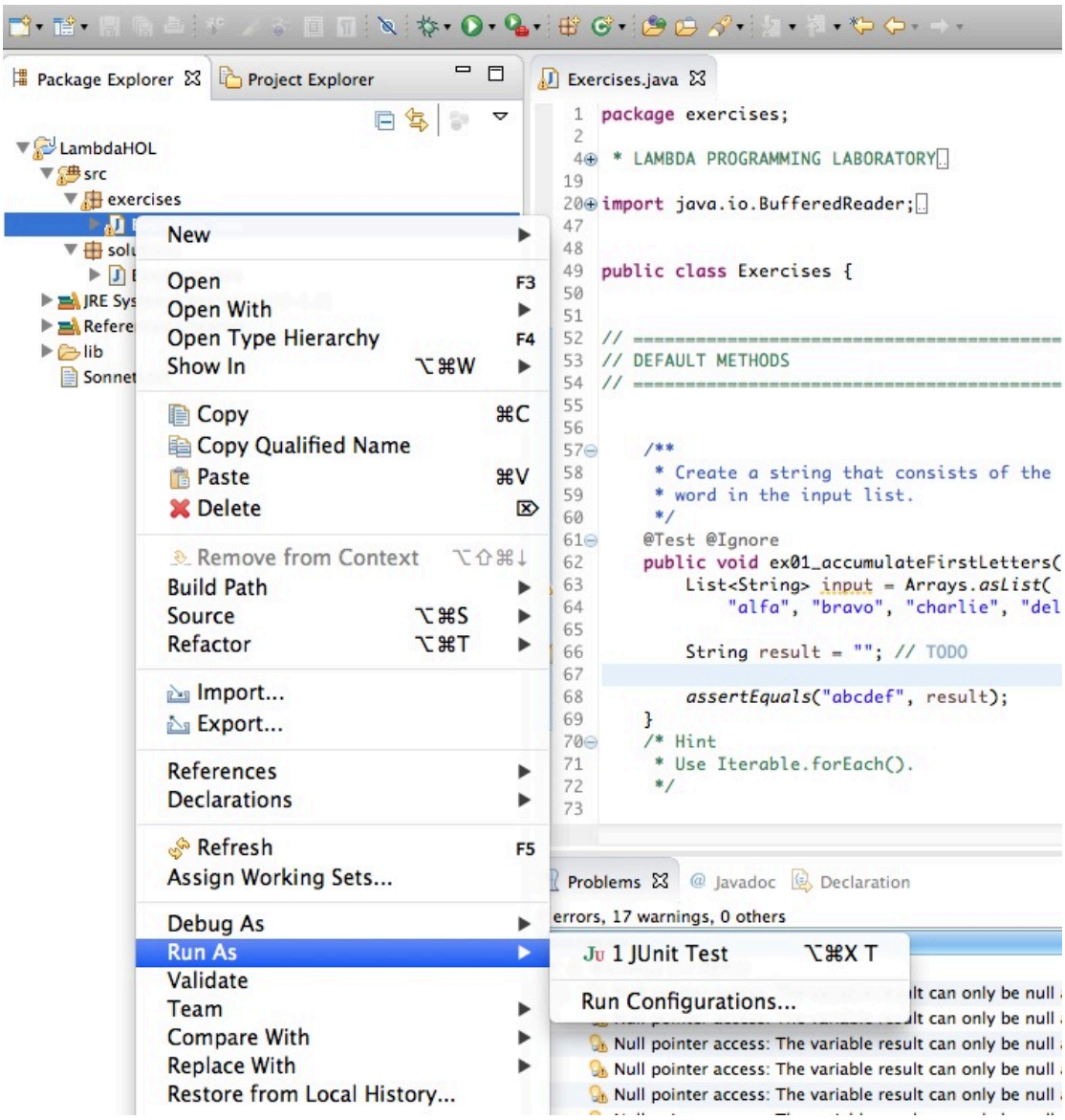
In addition to the Streams APIs there are some new methods in the Collections API that can use Lambda expressions and which will be used in the exercises:

- **`Collection.removeIf()`**: Removes all of the elements of this collection that satisfy the given `Predicate`.
- **`List.replaceAll()`**: Replaces each element of this list with the result of applying the `UnaryOperator` to that element.

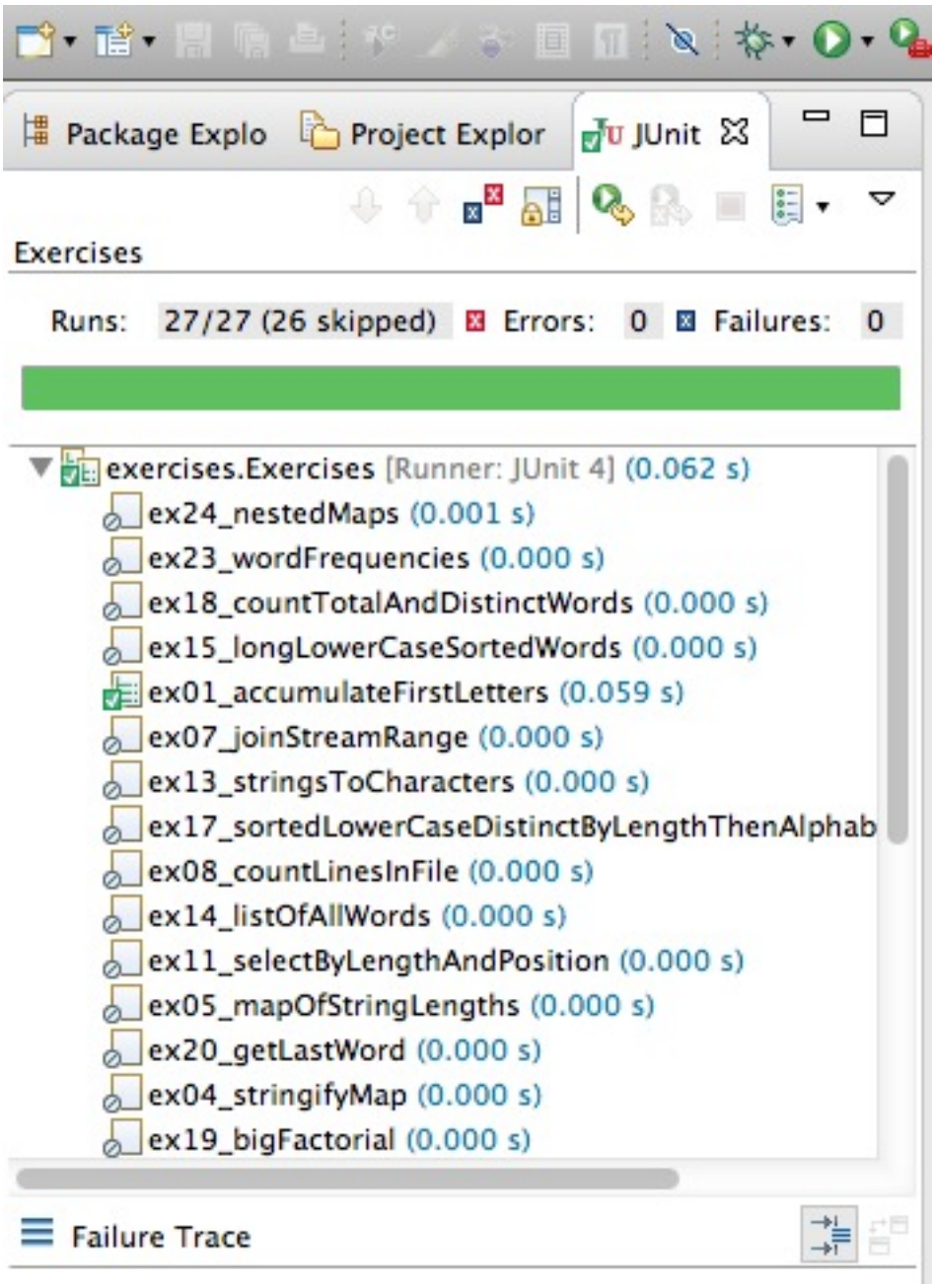
Running The Exercises

The exercises are provided as a set of unit tests using JUnit within the IDE.

1. Unzip the Lambda_HOL.zip file to your Eclipse workspace.
2. Start Eclipse and open the LambdaHOL project from your workspace.
3. There are two packages in the project, `exercises` and `solutions`. Each has a single class file, `Exercises.java`. Use the one in the `Exercises` package to construct your solutions to the problems. Refer to the solutions version if you get really stuck and need to see what the answer is.
4. Each exercise is contained in its own method. The comments above the method describe what you need to do to get the test to pass.
5. When you have written your solution ensure that the `@Ignore` annotation has been removed from the method.
6. Right click on the `Exercises.java` file in the package explorer and select `Run As` from the menu, then `JUnit Test` (see screen shot below)



7. A JUnit tab will open on the left and you will be able to see the results of the tests you've run along with any error messages, in the unlikely event that you don't get it right first time.



8. Repeat for as many exercises as you have time for. Each exercise includes one or more hint to help get you started. It is very likely you will need to refer to the API documentation frequently to understand the APIs you need to use.