Java 8 new features

Juan Hernández

Dec 1st 2015

Introduction

In this session we will do an introduction to the new features introduced by Java 8 for functional programming: λ-expressions and the streams API.

Ops!

Looks like audio isn't working in my laptop, I will use the phone.
Please join this BlueJeans meeting:

https://bluejeans.com/7539914530

Agenda

These are the items that we will cover in this session:

- Motivation
- λ-expressions
- The streams API

Motivation

Why are λ-expressions and the streams API needed?

- Java lacks a mechanism to express processing of collections of objects in a declarative/parallel way.
- There are sophisticated mechanism for concurrency, like the `java.util.concurrent` package, or the fork/join framework, but they aren't simple enough to use them every day.
- Developers often resort to traditional serial iteration, which is hard to parallelize and doesn't take advantage of modern multi-core architectures.

An example

This is the typical code that we most of us would write to find the subset of virtual machines that are up:

```
List<Vm> all = ...;
List<Vm> up = new ArrayList<>();
for (Vm vm : all) {
  if (vm.getStatus() == VmStatus.UP) {
    up.add(vm);
  }
}
```

So, what is the problem with this, isn't it perfectly good?

First problem

The first problem is the loop:

```
List<Vm> all = ...;
for (Vm vm : all) {
  if (vm.getStatus() == VmStatus.UP) {
    up.add(vm);
  }
}
```

This loop is inherently serial, it is almost impossible to parallelize, even if you use a very smart hot-spot virtual machine.

Second problem

The second problem is the use of the a local variable to accumulate the results:

```
List<Vm> all = ...;
List<Vm> up = new ArrayList<>();
for (Vm vm : all) {
  if (vm.getStatus() == VmStatus.UP) {
    up.add(vm);
  }
}
```

Even if you explicitly split the original list and hand it over to different threads, you will have a hard time to merge the results, as a simple list like this can't be safely accessed from multiple threads.

How is this improved with the streams API?

The streams API is the mechanism to move the iteration from your code to the library, where parellelizing it is more feasible:

```
List<Vm> all = ...;
List<Vm> up = all.stream().filter(
    new Predicate<Vm>() {
        public boolean test(Vm vm) {
            return vm.getStatus() == VmStatus.UP;
        }
    }
}).collect(toList());
```

But the way to express the business logic is still verbose. There is where λ-expressions come handy.

How is this improved with λ-expressions?

The λ-expressions are the mechanism to express the business logic in a concise way, and to pass this logic to the streams API:

```
List<Vm> all = ...;
List<Vm> up = all.stream()
  .filter(vm -> vm.getStatus() == VmStatus.UP)
  .collect(toList());
```

The 11th letter of the Greek alphabet

The Java λ operator

Agenda

- Motivation
- λ-expressions
  - λ-expression syntax
  - Functional interfaces
  - Access to members, variables and parameters
  - λ-expression aren't objects
  - Method references
- The streams API

λ-expression syntax

λ-expressions are anonymous *functions*:

```
( parameters ) -> { body }
```

- The parameters and the body are separated by the new λ-operator: `->`
- `parameters` is a list of parameter declarations separated by commas, like in the declaration of a method.
- `body` is a list of statements, like in the body of a method.
- They are *not* methods, but *functions*, more on that later.

λ-expression syntax

```
() -> {}
```

Equivalent to:

```
void function() {
}
```

Pretty useless, but shows a couple of interesting syntactic properties:

- Parenthesis around the parameters are mandatory when there are no parameters.
- Curly braces around the body are are mandatory when there are no statements.

λ-expression syntax

```
int x -> x + 1
```

Equivalent to:

```
int function() {
    return x + 1;
}
```

- The parenthesis around the parameters are optional when there is only one parameter.
- When the body contains exactly one statement, and that statement is an expression, then the curly braces, the `return` keyworkd and the semicolon at the end of the statement can be ommited.

λ-expression syntax

```
(String to) -> System.out.println("Hello " + to + "!")
```

Equivalent to:

```
void function(String to) {
  System.out.println("Hello " + to + "!");
}
```

- The body doesn't need to return a value.
- When the body contains only one statement, the curly braces and the semicolon at the end of the statement can be ommited.

λ-expression syntax

```
(int x, int y) -> { return x + y; }
```

Equivalent to:

```
int function(int x, int y) {
  return x + y;
}
```

- Parenthesis around the parametes are mandatory when there are more than one parameter.
- Curly braces, and the `return` keyword, can be used even if they aren't needed, but then the body must use the usual method body syntax, including the the semicolon at the end of the `return` statement.

λ-expression syntax

```
Vm vm -> {
  int gib = 10;
  vm.setMemory(gib * 1024 * 1024);
  vm.setDescription("With " + gib + " GiB");
  return vm;
}
```

Equivalent to:

```
Vm function(Vm vm) {
  int gib = 10;
  vm.setMemory(gib * 1024 * 1024);
  vm.setDescription("With " + gib + " GiB");
  return vm;
}
```

- The body can declare local variables, and have multiple exceptions, like the body of a method.

λ-expression syntax

The types of the parameters can often be ommited, the compiler can infer their types from the sorrounding environmnet. Lets assume that you have the following method:

```
T sort(List<T> list, Comparator<T> comparator)
```

You can call it passing a λ-expression to sort a list of virtual machines by their amount of RAM:

```
List<Vm> vms = ...;
sort(vms, (Vm x, Vm y) -> x.getMemory() - y.getMemory());
```

In this context the compiler can infer that the type of the `x` and `y` parameters is `Vm`, so the type can be ommited:

```
List<Vm> vms = ...;
sort(vms, (x, y) -> x.getMemory() - y.getMemory());
```

Functional interfaces

- Functional interfaces are the types of λ-expressions: a λ-expression can be used in any place where a functional interface can be used.
- A functional interface is an interface with exactly one abstract method.
  - Static methods don't count.
  - Default methods don't count.
  - Methods inherited from `java.lang.Object` (`toString`, `equals`, etc) don't count.
- Can optionally be marked with `@FunctionalInterface`.

Functional interfaces

Many existing interfaces are already functional. For example, the well known `Runnable` and `Callable` interfaces are functional:

```
public interface Runnable {
  void run();
}
```

```
public interface Callable<V> {
  V call() throws Exception;
}
```

Note that these interfaces don't need to extend any special base, or any special annotation, and that the signature of the method isn't relevant. The only relevant fact is that they have exactly one abstract method.

Functional interfaces

You can create your own functional interfaces (some may already be):

```
public interface VmHandler {
  void handle(Vm vm);
}
```

If you create your on functional interfaces it is good practice to mark them with the `@FunctionalInterface` annotation, so that the compiler will check that they actually are functional:

```
@FunctionalInterface
public interface VmHandler {
  void handle(Vm vm);
}
```

This is similar to `@Override`, and it isn't required.

Functional interfaces

The most interesting functional interfaces have been introduced by Java 8 in the `java.util.function` package, intended to work together with other Java class libraries:

```
package java.util.function;

@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
```

We will see them later, as they make more sense combined with the streams API.

Functional interfaces

λ-expressions can be used in any place where a functional interface can be used. For example, the traditional way to create a thread is this:

```
Thread thread = new Thread(
  new Runnable() {
    public void run() {
      doSomething();
    }
  }
);
```

As `Runnable` is a functional interface you can a λ-expression instead:

```
Thread thread = new Thread(() -> doSomething());
```

Access to members

λ-expressions can use members from their sorrounding environment:

```
public class VmHandler {
  private Vm vm;

  public void startThread() {
    Thread thread = new Thread(() -> {
      System.out.println(vm.getName());
    });
    thread.start();
  }
}
```

The rules are the same than control acess to members from anonymous inner classes.

Access to local variables

λ-expressions can use local variables from their sorrounding environment:

```
public class VmHandler {
  public void startThread() {
    Vm vm = ...;
    public Thread thread = new Thread(() -> {
      System.out.println(vm.getName());
    });
    thread.start();
  }
}
```

Like anonymous inner classes, λ-expressions can only use *final* local variables. But syntax has been relaxed in Java 8: the `final` keyword isn't required. I is enough if the compiler can determine that the variable is actually final. This kind of local variables are called *effectively final*.

Access to method parameters

Same for method parameters, λ-expressions can use them if they are effectively final, no `final` keyword required:

```
public class VmHandler {
  public void startThread(Vm vm) {
    public Thread thread = new Thread(() -> {
      System.out.println(vm.getName());
    });
    thread.start();
  }
}
```

λ-expressions aren't objects

λ-expression are anonymous functions, they aren't objects. An important consequence of that is that the `this` keyword has different semantics:

```
public class Test {

  public void go() {

    Runnable lambda = () -> {
      System.out.println("lambda: " + this.getClass());
    };

    Runnable inner = new Runnable() {
      public void run() {
        System.out.println("inner: " + this.getClass());
      }
    };

    lambda.run();
    inner.run();
  }

}
```

λ-expressions aren't objects

The result of compiling the previous class are two `.class` files, one for `Test.class` file for the main class, and one `Test$1.class` file for the anonymous inner class:

```
$ javac Test.java
$ ls *.class
Test.class Test$1.class
```

But no `.class` file has been generated for the λ-expression, as it isn't an object.

Running the program generates the following output:

```
$ java Test
lambda: class Test
inner: class Test$1
```

Method references

There are many situations where a λ-expressions contains just a method call. For example, you may need very often to get the name of a virtual machine, and for that you can use a λ-expression like this:

```
vm -> vm.getName()
```

This can be replaced by a method reference:

```
Vm::getName
```

The left hand side, before the `::` is called the *target* of the reference. The right hand side is just the name of the method.

Method references: target

When the target of the method reference is a class name, and the method isn't static, the object corresponds to the first parameter of the expression, and the method paremeters correspond to additional parameters of the expression. So this method reference:

```
Vm::setName
```

Is equivalent to this λ-expression:

```
(Vm vm, String name) -> vm.setName(name)
```

Method references: target

When the target of the method reference is a class name, and the method is static, then all the parameters of the method correspond to parameters of the expression:

```
VmHandler::isUpdateValid
```

Is equivalent to this λ-expression:

```
(VmStatic source, VmStatic destination, VMStatus status) ->
  VmStatic.isUPdateValid(source, destination, status)
```

Method references: target

When the target of the method reference is a value, that value doesn't correspond to any parameter of the expression, rather it is taken from the environment. The rest of the parameters of the method correspond to parametrs of the expression:

```
vm::setName
```

Is equivalent to this λ-expression:

```
String name -> vm.setName(name)
```

Method references: constructors

Method references can be also used for constructors, in that case the name of the method is new:

```
Vm::new
```

Is equivalent to:

```
() -> new Vm()
```

Method references: constructors

Works also for arrays:

```
Vm[]::new
```

Is equivalent to:

```
int length -> new Vm[length];
```

λ-expressions summary

- Basic syntax:

```
( parameters ) -> { body }
```

- A functional interface is one with exactly one abstract method.
- Access rules almost identical to annonymous inner classes, except `this`.
- Method references simplify many common λ-expressions.

The streams API

Agenda

- Motivation
- λ-expressions
- The streams API
  - Overview
  - Stream sources
  - Intermediate operations
  - Terminal operations
  - Optional

Overview

- Streams are a mechanism to express in a declarative way the processing of collections of objects.
- The iteration moves from the application to the library.
- The library can optimize the processing in many sophisticated ways, like paralellization and lazy processing.

Overview

Streams have three components:

- One source.
- Zero or more intermediate operations.
- One terminal operation.

```
List<Vm> vms = ...;                          // <- Source
long totalUpMemory = vms.stream()
  .filter(vm -> vm.getStatus() == UP)  // <- Intermediate
  .mapToLong(vm -> vm.getMemory())     // <- Intermediate
  .sum();                                    // <- Terminal
```

Stream sources: collections

The most typical sources of streams are collections. The `java.util.Collection` interface has now the following methods:

```
Stream<E> stream()
```

- Generates a stream that processes the elements of the collection sequentially.

```
Stream<E> parallelStream()
```

- Generates a stream that uses the fork/join framework to process the elements of the collection in parallel.

Stream sources: arrays

The `java.util.Arrays` class has methods to generate streams from array of objects:

```
<T> Stream<T> stream(T[] array)
```

It also has methods to generate streams from arrays of primitive types:

```
IntStream stream(int[] array)
LongStream stream(long[] array)
DoubleStream stream(double[] array)
```

Stream sources: other

Many other classes can also provide streams. For example the `java.io.BufferedReader` class has a method that generates a stream of lines:

```
Stream<String> lines()
```

The `java.nio.file.Files` class has a method that generates a stream of files from a directory:

```
Stream<Path> list(Path dir)
```

In general, every place that used to return a collection or array is now a candidate for returning a stream, try to use them.

Stream sources: static methods

The `java.util.streams.Stream` interface contains many static methods that generate useful streams. For example:

```
<T> Stream<T> empty()
```

- Generates an empty stream.

```
<T> Stream<T> of(T t)
```

- Generates an stream that process one single element.

```
<T> Stream<T> of(T... values)
```

- Generates an stream that process a fixed set of elements.

And many others, explore it.

Intermediate operations

- Intermediate operations are added to the stream with methods like `filter` and `map`.
- They are *remembered* by the stream, but not performed inmediately.
- They may be re-ordered, fused or even optimized away by the library, if that doesn't change the semantics of the complete operation.

Intermediate operations: filter

- The `filter` operation transform the stream so that it will only process the objects that satisfy a given predicate.
- The type of the argument of the `filter` method is the `java.util.functions.Predicate` functional interface:

```
interface Predicate<T> {
  boolean test(T t);
}
```

- Typically expressed using a λ-expression:

```
vms.stream()
  .filter(vm -> vm.getStatus() == UP)
```

Intermediate operations: map

- The `map` operation applies a function to the each object, transforming it into a different object.
- The type of the argument of the `map` method is the `java.util.functions.Function` functional interface:

```
interface Function<T, R> {
  R apply(T t);
}
```

- Typically expressed using a λ-expression:

```
vms.stream()
  .map(vm -> vm.getName())
```

Intermediate operations: map

There are specialized versions of `map` and `Function` that work with primitive types, for example for `long`:

```
interface ToLongFunction<R> {
  long applyAsLong(T value);
}
```

```
List<Vm> vms = ...;
vms.stream()
  .mapToLong(vm -> vm.getMemory())
```

These perform much better than boxing and unboxing primitive types.

Intermediate operations: skip and limit

The `skip` operation is used to skip processing of the first *n* elements of the stream:

```
vms.stream()
  .skip(1)   // Don't process the first element
```

The `limit` operation is used to process only the first *n* element:

```
vms.stream()
  .limit(10)  // Process only the first 10 elements
```

And they can be combined:

```
vms.stream()
  .skip(1)   // Skip the first element
  .limit(1)  // Process only one element, the second
```

Intermediate operations: sorting

The `sorted` operation sorts the elements of the stream before processing them, either using their natural order:

```
vms.stream()
  .sorted()  // Uses the natural sorting of "Vm"
```

Or a custom comparator, typically expressed as a λ-expression:

```
vms.stream()
  .sorted((x, y) -> x.getMemory() - y.getMemory())
```

Terminal operations

Terminal operations trigger the processing of the elements of the stream, using the intermediate operations that have been previously added and generating results.

Terminal operations: findFirst

Returns the first element of the stream:

```
Optional<Vm> vm = vms.stream()
  .findFirst();
```

The result is an `Optional` container, will see it later.

It is typically combined with `filter`, to find the first element that satisfies a given condition:

```
Optional<Vm> vm = vms.stream()
  .filter(vm -> vm.getState() == UP)
  .findFirst();
```

Terminal operations: findAny

Returns any element of the stream:

```
Optional<Vm> vm = vms.stream()
  .findAny();
```

Typically combined with `filter`, to find any element that satisfies a given condition:

```
Optional<Vm> vm = vms.stream()
  .filter(vm -> vm.getState() == UP)
  .findAny();
```

The difference between `findAny` and `findFirst` is the order, `findAny` doesn't guarantee any order, and it works specially well with parallel streams.

Terminal operations: allMatch

Processes the elements of the stream and checks that all of them satisfy the given predicate:

```
boolean allUp = vms.stream()
  .allMatch(vm -> vm.getStatus() == UP);
```

Terminal operations: anyMatch

Processes the elements of the stream and checks if at least one of them satisfies the given predicate:

```
boolean atLeastOneUp = vms.stream()
  .anyMatch(vm -> vm.getStatus() == UP);
```

Like `findAny` this works specially well with parallel streams.

Terminal operations: noneMatch

Processes the elements of the stream and checks if note of them satisfies the given predicate:

```
boolean noneUp = vms.stream()
  .noneMatch(vm -> vm.getStatus() == UP);
```

Terminal operations: toArray

Returns an array containing all the elements of the stream:

```
Sring[] allNames = (String[]) vms.stream()
  .map(vm -> vm.getName())
  .toArray();
```

Best combined with a method reference to avoid the explicit cast:

```
Sring[] allNames = vms.stream()
  .map(vm -> vm.getName())
  .toArray(String[]::new);
```

Terminal operations: collect

The `collect` operations uses an instance of the non-functional `java.util.streams.Collector` interface to compute a result. We aren't going to go into the details of this interface, rather will see some typical existing collectors.

Terminal operations: toList

The `toList` collector creates a new list and adds all the elements of the stream:

```
static import java.util.streams.Collectors.toList;

List<String> allNames = vms.stream()
  .map(Vm::getName)
  .collect(toList());
```

It is common practice use static imports for this methods, but not mandatory.

Terminal operations: toSet

The `toSet` collector creates a new set and adds all the elements of the stream:

```
Set<String> allNames = vms.stream()
  .map(Vm::getName)
  .collect(toSet());
```

Terminal operations: toMap

The `toMap` collector creates a new map and populates it with keys and values extracted from the elements of the stream, using functions that you pass as parameters:

```
Map<String, Vm> byName = vms.stream()
  .collect(toMap(
    vm -> vm.getKey(), // This extracts the key
    vm -> vm           // This extracts the value
  ));
```

Method references are handy here, and so is the `identity` static method of the `Function` interface:

```
Map<String, Vm> byName = vms.stream()
  .collect(toMap(Vm::getName, identity());
```

Terminal operations: joining

The `Collectors` class contains a set of `joining` methods for generating strings. For example to create a comma separated list without having to deal with the typical problem of not adding the comma after the last element:

```
String withCommas = vms.stream()
  .map(Vm::getName)
  .collect(joining(", "));
```

Also possible to specify a prefix and a suffix:

```
String withCommasAndBrackets = vms.stream()
  .map(Vm::getName)
  .collect(joining(", ", "[", "]"));
```

Terminal operations: numeric

Streams also support terminal operations that produce numeric results, like `count`, `average` and `sum`:

```
long upCount = vms.stream()
  .filter(vm -> vm.getStatus() == UP)
  .count();
```

```
long averageUpMemory = vms.stream()
  .filter(vm -> vm.getStatus() == UP)
  .mapToLong(vm -> vm.getMemory())
  .average();
```

```
long totalUpMemory = vms.stream()
  .filter(vm -> vm.getStatus() == UP)
  .mapToLong(vm -> vm.getMemory())
  .sum();
```

Terminal operations: forEach

The `forEach` iterates all the elements of the stream of for each of them it perform an action. This action is passed as an instance of the `Consumer` functional interface:

```
public interface Consumer<T> {
  void accept(T t);
}
```

Usually expressed using a λ-expression:

```
vms.stream()
  .forEach(vm -> vm.start());
```

Optional

When operating on a stream chances are that it will be empty. To avoid null pointer exceptions the methods that return an element, like `findFirst` or `findAny`, return an instance of `Optional`:

```
Optional<Vm> up = vms.stream()
  .filter(vm -> vm.getStatus() == UP);
  .findFirst();
```

The presence of the content can be checked with the `isPresent` method, and the content can be retrieved with the `get` method:

```
Optional<Vm> up = ...;
if (up.isPresent()) {
  System.out.println(up.get().getName());
}
```

Optional: ifPresent

But using `get` isn't the best thing in the world, as it throws the unchecked `NoSuchElementException` exception when the optional is empty. If you aren't careful you may find yourself dealing with NSEE instead of NPE, not that an improvement.

Instead of that you can use the `ifPresent` method, which receives a `Consumer`, tipically expressed as a λ-expression:

```
vms.stream()
  .filter(vm -> vm.getStatus() == UP)
  .findFirst()
  .ifPresent(vm -> System.out.println(vm.getName()));
```

This avoids the explicit `null` check, quite an improvement.

Optional: map and filter

The `Optional` class works like a small stream, limited to zero or one element. This means that it also has the `map` and filter operations:

```
vms.stream()
  .filter(vm -> vm.getStatus() == UP)
  .findFirst()
  .map(Vm::getName)
  .ifPresent(System.out::println);
```

Optional: orElse and orElseThrow

The `orElse` and `orElseThrow` operations are very useful to integrate new code with existing code that expect `null` or an exception when the result of a stream operation is empty:

```
return vms.stream()
  .filter(vm -> vm.getStatus() == UP)
  .findFirst()
  .orElse(null);
```

```
return vms.stream()
  .filter(vm -> vm.getStatus() == UP)
  .findFirst()
  .orElseThrow(NoSuchElementException::new);
```

Summary

- Streams are the mechanism to express declaratively the processing of collections of objects.
- Streams are created from a source, and have intermediate and terminal operations.
- The `Optional` class is like a mini-stream, with zero or one element.

What else?

This was just an introduction, covering the basic concepts. There are more advanced topics like collectors, the *reduce* operation, parallel streams and *spliterators*. If you want to go deeper I'd suggest you start by reading the Javadoc, starting with the `java.util.stream` package.

Thanks!