Lambda Expressions and Java 8 Streams

Jan Trienes, Thijs Dorssers, Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

March 13, 2018

Contents of this talk

Lambda Expression

Definition

Lambda expressions are basic ingredients of so the called *Lambda calculus* which is a formal language for describing functions and their definition.
It makes it possible to notate functions without the need to give them a name.

For Java programmers

Lambda expressions express a piece of functionality.

```
(int x) -> { return 2x+1; }
```

Why Lambdas

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times.
We have seen this before with anonymous classes:

Anonymous class: a button callback

```
button.setOnAction(
  new EventHandler<ActionEvent> () {
    public void handle(ActionEvent event) {
      System.out.println("Thanks for clicking");
  }
});
```

Syntax

Properties of a lambda:

- ✓ Parameter List
- ✓ Function Body
- ✓ Return Type
- ✓ Exceptions
- ✗ Name
- ✗ Javadoc

Lambda Syntax Java

```
(int x) -> { return 2x+1; }
```

Syntax Explanation

- The parameter list is placed before the `->` in `()` (parenthesis). The lambda body follows afterwards in {} (braces).
- Although not obvious, a lambda expression can return a value or throw an exception.
- Return type and exceptions are inferred by the compiler.
- The only thing that a lambda lacks, is a name. Therefore a lambda expression is sometimes called **anonymous method**.

Intermezzo: Signature and Shape

- The **signature** of a method is defined as the **name** plus the **argument list**.
  - The return type is ignored.
  - Other keywords such as `public`, `static`, `final`,`synchronized` are ignored in the signature as are `throws` clauses.
  - The parameter *names* are also ignored.
- A class cannot not have multiple methods with the same signature. So you need to vary:

  name which introduces a new method, which can have different effect with the same parameters. E.g. `Fraction.mul(Fraction f)` vs `Fraction.mul(Fraction f)`

  parameter list which we call **overloading**
- Generic Type parameters do NOT count in the signature definition, because of type erasure.

Intermezzo: Signature and Shape continued.

- We use the word **shape**[1] to describe other aspects of methods.
- The **shape** is the **parameter list**, **return value**, the **throws clause**, and the **generic types of the parameters**.
- It differs from the signature.
- A shape can be reused in a class, as long as the name of the method is different.
  - This allows the use of so called method references to methods of the same class, fitting the same shape, e.g. event listeners that trigger different methods of the same class.
- The shape is the essential definition that is provided in a (functional) `interface`.
- To be able to declare a λ expression, the used shape must exist.
- The shape is used by the compiler to infer (make a best guess) on what method to choose from the possible candidates.
- Quite a few shapes are already provided in the `java.util.function` package.

[1] Shape is the word we use in Venlo until there is a better and accepted word in the industry or literature.

Example 1

The definition of a lambda expression via a functional interface.

```
@FunctionalInterface
interface Calculator {

    /**
     * Compute a value from the inputs.
     *
     * @param a first input
     * @param b second input
     * @return result of the computation
     */
    int calculate( int a, int b );
}
```

Who is SAM again?

Example 1, continued

```
1   public static void main( String[] args ) {
2       ReCalc reCalc = new ReCalc();
3       int result;
4       // with anonymous inner class
5       Calculator cAnon = new Calculator() {
6
7           @Override
8           public int calculate( int a, int b ) {
9               return a + b;
10          }
11      };
12      result = reCalc.calculate( cAnon, 10, 32 );
13      System.out.println( result );
14      // with lamba expression
15      result = reCalc.calculate( ( x, y ) -> x + y, 12, 30 );
16      System.out.println( result );
17      you can use an expession as 'data' too.
18      Calculator cLambda = ( x, y ) -> x + y;
19      result = reCalc.calculate( cLambda, 12, 30 );
20      System.out.println( result );
21  }
```

Example 2

Define a method to filter some input to a new list.

```
1     List<Person> filter( List<Person> persons,
2             Predicate<? super Person> filter ) {
3
4         List<Person> result = new ArrayList();
5
6         for ( Person p : persons ) { // see also stream version
7             if ( filter.test( p ) ) {
8                 result.add( p );
9             }
10        }
11
12        return result;
13    }
14
15    public static void main( String[] args ) {
16        LambdaShowcase sc = new LambdaShowcase();
17        printList( "all", somePersons() );
18
19        List<Person> ofAgeLambdaMethod = sc.filter( somePersons(),
20                p -> p.isFullAged() );
21        printList( "ofAgeAlt = " , ofAgeLambdaMethod );
22
23        List<Person> ofAgeLambdaDiys = sc.filter( somePersons(),
```

Notice line 2: See Java 8 `java.util.function.Predicate`

Syntax Short Forms

Several syntax variations exist:

```
1  // parameter type, return statement
2  filter( personList, (Person p) -> { return p.isFullAged();});
3  // infered parameter type and return type.
4  filter( personList, p -> p.isFullAged());
5  //the last one is a method reference on p
6  filter( personList, p::isFullAged );
7  // You can also write Person::isFullAged, which unambiguously
8  // resolves to the same instance (as in non static) method.
9  filter( personList, Person::isFullAged );
10 //the one below is a static method reference...
11 //... in class Person.
12 filter( personList, Person::hasPrimeAge );
```

Notice that a **method signature** does **NOT** include the `static` modifier, so a method signature refers to *either* a static *or* an instance method and can be inferred unambiguously.

Method Reference and Lambda

Method Ref type	Example	Lambda Equivalent
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then= Instant.now();` `t -> then.isAfter();`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>();`
Array Constructor	`int []:: new`	`(len) -> new int[len];`

From Effective Java 3rd ed.

Examples from java.util.function

One should prefer these functions over home-grown functional interfaces where possible.

Interface	Function Shape	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1,T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

From Effective Java 3rd ed.

Ad-hoc Functionality

Ad hoc implementation

Lambda expressions are implemented *"ad hoc"*, right when and where they are needed. This is similar to anonymous inner-classes.

Code as data

Both lambda expressions and anonymous inner-classes are *passed as arguments* to a function which applies them. This is known as *code as data*.

Are related to:

- delegates in Swift, objective-C (Apple) and C# (dot Net)
- call back functions in C/C++

Lambda vs. Anonymous inner-class

Without lambdas:

Anonymous inner-class

```
filter(personList, new PersonFilter() {

    @Override
    public boolean accept(Person p) {
        return p.isFullAged();
    }

});
```

Same with lambdas:

Lambda expression

```
filter(personList, p -> p.isFullAged());
```

Lambda vs. Anonymous inner-class

- Lambda expressions can be used where in places where interface implementing anonymous inner-classes have to implement only one method. (SAM).
- They reduce the syntactical overhead/clutter or ceremony.
- Object creation is no longer needed on programmers side. In Java methods always belong to an object. This is true for lambda functions (anonymous methods) too. Here also an object must be created.[2]

[2] which is taken care of by the compiler

Things you CANNOT do with Lambda

So you might still want to use an (anon) inner class.

- From a code block, like and anonymous inner class you can read and modify any local variabl in scope; from an lambda you cal onlye *read* final or effectively final variables, and you cannot modify any local variables.
- From a code block you can return from the enclosing method, `break` or `continue` an enclosing loop, or throw a checked exception declared in the throws clause of said method; from a lambda you cannot do any of these actions.

source: Effective Java 3rd ed.

Where lambdas shine

- Uniformly transform sequences of elements
- Filter sequences of elemements
- Combine sequeneces of elements using a single operation(eg. add the, concatenate them, compute min or max)
- Accumulate sequences of elements into collections, with grouping as possiblity thrown in.
- Search a sequence of elements for (any, first or all) element(s) sattisfying some criterion.

source: Effective Java 3rd ed.

Representation of Lambda Expressions

Function → Object

When a lambda expression is used, we define a function. The receiving part uses it as an object.

c is the receiving object

```
public ReCalc(){}

public int calculate(Calculator c, int n) {
  return c.calculate(n);
}
```

Internal representation

As everything else in Java, a lambda expression is a regular object, which has an address and a type[a]

[a] An sub-type of the shape defining interface.

Functional Interfaces

- As mentioned before, a lambda expression is of a certain type. This type is a subtype of the so called *target type*.
- Target types of a lambda expression are declared in Java as a `@FunctionalInterface`, like:

```
@FunctionalInterface
interface Calculator {
  int calculate(int n);
}
```

- A functional interface is an interface with only *one* **abstract** method.
- There are plenty of functional interfaces in Java. For instance: `Runnable`, `Callable`, `ActionListener`, `Comparator`...
- Lambdas are backwards compatible with an interface with only one single abstract method, also know as SAM for Single Abstract Method.
- Many functional interfaces are declared in the package `java.util.function`

Type Inference

- In the example, we never specified that the lambda expression is of type `Calculator`.
- The compiler resolves this with ***type inference***.
- The compiler inspects the context in which the lambda expression is created and figures out which type is needed.
- The given lambda expression is validated against this type as in *(does the shape match?)*.
- In Venlo we call the signature *minus* name *plus* return type and throws clause the **shape** of a method.
  If the shape matches, the compiler creates a sub-type of the target-type.

Functional interface and Shape

```
interface BiFunction<T,U,R> {
    R apply(T t, U u);
}
// fits to shape
(T t, U u) -> expression of type R
```

Rules and shortcuts.

- If there is no parameter, a set of empty parenthesis is required.
- If there is no return value (void shape), then curly braces are required, otherwise the type of the return value is equals the type of the expression.
- If there is exactly one parameter, you can have parenthesis plus type plus parameter name or just parameter name and drop the parameters and parameter type .
- with multiple parameters, parenthesis are required, with or without or parameter types. With types it is all or nothing.

Variable Binding and Scoping

Access to enclosing context

Occasionally an anonymous inner-class or a lambda expression needs access to variables in the enclosing context. The mechanism to support this is called *variable binding*.

Scoping

A anonymous inner-class and a lambda expressions differ in their scoping.

Variable Binding

Consider the following example:

Variable Binding

```
void method() {
    Work work = new Work();

    Thread t = new Thread(() -> {
        work.process();
    });
}
```

- The lambda expression needs access to a variable defined in the enclosing context.
- This variable is now bound to that expression. *(implicitly final)*

Scoping

Anonymous Inner-class vs. Lambda Expressions

Own scope vs. lexical scope

```
1  {
2    int n = 0;
3
4    Thread t = new Thread(new Runnable() {
5       @Override
6       public void run() {
7         int n = 0; //this is fine
8       }
9    });
10
11   Thread t2 = new Thread(() -> {
12      int n; //compiler complains, n already defined
13   });
14 }
```

Internal vs. External Iteration

- The collections framework now supports internal iteration
- This is a clear separation of concerns. The "how" and the "what" is encapsulated and independent

Consider the difference: both use `List<String> names = new ArrayList <>();`

External iteration

```
Iterator<String> it = names.iterator();
while(it.hasNext()) {
  System.out.println(it.next());
}
```

versus

Internal iteration

```
names.forEach(name -> System.out.println(name));
```

Notice: the for-each loop is the same as external iteration.

Internal vs. External Iteration

External Iteration

The client takes full control over the iteration process and the actions that are applied to each item.

Internal Iteration

The sequence (produced by the collection) itself determines how elements in a sequence are accessed and the user defines the actions applied the items.

Streams API

New functionality for Collections framework

- Support for parallel bulk operations
- Heavily uses the stream concept (like in filter())
- Better separation of “what” or “how” based on lambda expressions.

What is a stream?

A stream seems in some way similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

- A stream does not store any elements. It is more or less a n (active) bystander. The elements are 'produced' by an underlying collection or generated on demand.
- Stream operations do not mutate their sources. Instead, the source typically returns a new stream for further use.
- Stream operations are **lazy** when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of counting them all, then the filter will stop filtering after the fifth match.

Operations on Streams

Some of the new operations

`filter()` select. only those items, that match a predicate are contained. in the result. **I**

`map()` transform creates a new stream out of the results from the passed function. **I**

`distinct()` deduplicate returns a stream in which all items are distinct. Implies sorting of sorts. **I**

`count()` reduce returns the number of items in the stream. **T**

`forEach()` sic. applies a passed function to each item in the stream. **T**.

Terminal and Intermediate Operations

The previously mentioned operations belong to two categories:

Intermediate operation returns a stream, which can be used for further processing

Terminal operation returns a final result or void, which can not be used again

- **Intermediate** operations support a programming style which is known as *fluent programming*. It's the chaining of operations.
- The typical chain contains the operations *filter* (intermediate), *map* (intermediate), and *reduce*(terminal).
  - Reduce as in reduction: indikken, inkoken, einkochen.
  - reduce examples: count, take average, collect into list.

An Example using Streams

Goal: Find the names of all persons which are no older than 30 and which name ends with "daal".

Example of applying streams

```
void method() {
  List<Person> personList = new ArrayList<>();
  personList.add(new Person("John Doe", 15));
  personList.add(new Person("Jane Doe", 17));
  personList.add(new Person("Anne Modaal", 35));
  personList.add(new Person("Max Modaal", 19));
  personList.add(new Person("Erika Modaal", 28));

  personList
    .parallelStream() //supports parallel execution
    .filter(p -> p.age <= 30)
    .map(p::getName)
    .filter(s -> s.toLowerCase().endsWith("daal"))
    .forEach((String s) -> System.out.println(s));
}
```

Exercise See example above.
Find the stream implementation of the following: calculate the average age of these persons.

Additional Information

- Tutorial on Java 8 and lambda expressions by Angelika Langer and Klaus Kreft `http://angelikalanger.com/Lambdas/Lambdas.html`
- `java.util.stream`
- `java.util.function`