Java 8: Lambda Expressions

Ţurcan Olga
tosd2256@scs.ubbcluj.ro

Babeş-Bolyai University, Cluj-Napoca

Abstract

The purpose of this paper is to study the Java Lambda Expressions, which were introduced in Java 8 to support functional style programming. We will begin by explaining what a lambda expression is and analyzing its syntax. We will then explain how a lambda expression can be used based on some examples and outline what are the benefits of using lambdas in Java. Finally, we will show the usage of lambda expressions in the Java 8 APIs.

1 Introduction

In recent years, functional programming has become popular because of its suitability in concurrent, parallel, and event-driven programming. While some programming languages like LISP and Haskell have been fundamentally designed based on the principles of functional programming, other modern languages like C++ and Python have also adopted lambdas. Java did not want to be left behind, so it introduced lambda expressions to support functional programming, which can be mixed with its already popular object-oriented features to develop robust, concurrent, parallel programs.

2 What is a lambda expression?

The term "lambda expression" originates from Lambda calculus that uses the Greek letter lambda to denote a function abstraction. According to [4], "A lambda expression is an unnamed block of code representing a functionality that can be passed around like data". What this means is that a lambda expression can be assigned to a variable, passed around as a function argument or returned from a function. In Java 8, lambdas are constructs that represent instances of functional interfaces: interfaces with a single abstract method. They can be used

to achieve behavior parametrization. Behavior parametrization is not a new feature added in Java 8 - this was still possible in earlier versions using anonymous classes. However, as we will see later based on some code examples, lambdas use a very concise and clear syntax compared to anonymous classes to achieve the same result. Also, lambda expressions are the basis of other Java 8 features. For instance, the Stream API which offers new efficient ways to process data, is built on the idea of passing code to parameterize the behavior of its operations.

3 Syntax of Java Lambda Expressions

A lambda expression represents an anonymous function, thus its general syntax is similar to declaring a method. It consists of a list of parameters and a body, separated by an arrow:

```
(<LambdaParametersList>) ->  <LambdaBody>
```

The body can be a simple expression or a block of statements. Just like methods, the body of a lambda expression can declare local variables, throw exceptions and use break, return and continue statements. There are however, subtle differences between lambdas and methods, as described in [4]:

- A lambda expression does not have a name.
- A lambda expression does not have an explicit return type. The return type will be inferred by the compiler from its body and context of use.
- A lambda expression cannot have a throws clause.
- A lambda expression cannot be generic.

To further simplify the structure of a lambda expression, it is possible to omit the type of the parameters. This results in an implicitly-typed lambda expression, meaning that the compiler will infer the types of parameters from the context in which the lambda expression is used. If a single parameter is present, it is also possible to omit the paranthesis. If the body is composed of a single statement, braces can also be ommited. Listing 1 shows a few examples of syntactically valid lambda expressions.

```
// Takes 2 int parameters and returns their sum
(int a, int b) -> { return a + b; }

// Equivalent to the previous example, parameter types are ommited
(a, b) -> { return a + b; }
```

```
// Equivalent to the previous example, braces are ommited
(a, b) -> a + b

// Takes a String parameter and returns its length
(String str) -> { return str.length(); }

// Takes a parameter and prints it on the standard output
(msg) -> { System.out.println(msg); }

// Equivalent to the previous example, paranthesis and braces are ommited
msg -> System.out.println(msg)

// Takes no parameters and returns the string "noParam"
() -> "noParam"

// Takes a parameter and does nothing
param -> {}
```

Listing 1: Examples of Lambda Expressions

4 Functional Interfaces

We already mentioned that Java lambda expressions are instances of functional interfaces. A functional interface is nothing more than an interface with a single abstract method. Consider for example the well known `Comparator` interface presented in Listing 2. It declares a single abstract method `compare`, that takes two parameters of type T, and returns an int.

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Listing 2: The Comparator functional interface

We can create a lambda expression of this signature and assign it to a variable of type Comparator. Imagine we have a simple `Person` class with only 2 fields, `name` of type String and `age` of type int. We might want to create a Comparator that would compare instances of class Person based on the name. Using a lambda expression, this is one example of how it could look like:

```
Comparator<Person> comp = (a, b) -> a.getName().compareTo(b.getName());
```

In this case, it is clear that the lambda expression is of type Comparator<Person>, since it is assigned to the variable `comp` of that type. In practice however, lambdas are often written to be passed as parameters to methods and do not specify a type explicitely. In that case, the type of the lambda expression is inferred by the compiler based on the context in which it is used. This is known as **target typing**. The compiler analizes the context in which a lambda expression is used to determine if it is compatible with the expected target type.

Let's say we want to sort a list of Person objects based on their names. We could use the `sort` method from the `List` interface, which has a single parameter of type `Comparator`. Before Java 8, the best option would be to create an anonymous class that will implement the Comparator interface. However, since Java 8 we can use the lambda expression that we have written above to achieve our purpose. In that case, the compiler will analyze the lambda expression, and match it against the target type expected by the `sort` method. For the code to compile, the lambda expression must be compatible with its target type: number and type of parameters must match, as well as the return type. Example B of Listing 3 presents a correct implementation of passing a lambda expresions where a parameter of type `Comparator<Person>` is expected. In that case, the type of the lambda is still `Comparator<Person>`. Even if it is not explicitely specified, it is inferred by the compiler.

```
List<Person> persons = Arrays.asList(
    new Person("Maria", 22),
    new Person("Alexandru", 24));

// A. Sort using an anonymous class
persons.sort(new Comparator<Person>() {
    @Override
    public int compare(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }
});

// B. Sort using a lambda expression
persons.sort((a, b) -> a.getName().compareTo(b.getName()));
```

Listing 3: Sorting a List with a Comparator: A. using an anonymous class; B. using a lambda expression

Notice that example A from the same Listing 3 produces the same result as example B using an anonymous class. However, it is clear that the solution using lambda expressions is much more concise and clear.

Even if they produce the same result, anonymous classes and lambda expressions are not equivalent when translated to bytecode. As explained in [2], the compiler generates a new .class

file for each anonymous class, which may impact the startup performance of the application. Instead, the body of a lambda is converted to a method of the class in which it appears: a static method if the lambda uses no variables outside its body, or an instance method otherwise. This gives lambdas an advantage in performance compared to anonymous classes.

5 Behavior parametrization

In the previous section we have seen how to use lambda expressions for existing functional interfaces. This feature can also be used for custom defined functional interfaces to create behavior parametrization.

Suppose you are building an ecommerce platform and want to implement the option of filtering products based on various attributes: price, color and brand. The first thing that comes in mind could be to create 3 different methods, one for each attribute that would test if a product meets the desired parameters. But those 3 methods will probably have a common general structure and only a few lines that test the attribute values would differ. Using lambda expressions you can pass the logic for that few lines as a parameter, resulting in shorter and much cleaner code.

The solution for this problem is presented in Listing 4. We created the functional interface `ProductTester`, which has a single abstract method test. The `filetrProducts` method expects an instance of `ProductTester` as a parameter. It will be used when iterating the list of products to decide if the product should be displayed or filtered out. This way we can reuse the same piece of code and change its behavior by passing it into the method rather than hardcoding each case in a separate method.

```
class Product {
    private double price;
    private String color;
    private String brand;

    //getters and setters go here
}

interface ProductTester {
    boolean test(Product product);
}

public class ProductShop {
    public void filterProducts(List<Product> products, ProductTester tester) {
        for (Product p : products) {
            if (tester.test(p)) {
```

```
                System.out.println(p);
            }
        }
    }
}

... //Examples of usage

// Show products that have color white
productShop.filterProducts(products, p -> "white".equals(p.getColor()));

// Show products that have price between 200 and 500
productShop.filterProducts(products, p -> p.getPrice() > 200
        && p.getPrice() < 500);

// Show products that are of brand Apple and have price under 1000
productShop.filterProducts(products, p -> "Apple".equals(p.getBrand())
        && p.getPrice() < 1000);
```

Listing 4: Using a custom functional interface to create behavior parametrization

6 Standard functional interfaces

The functional interface we used in our previous example is a very simple one. It takes one parameter and returns a boolean value. This may be useful for a big number of use cases. As a result, Java 8 has added several standard functional interfaces. These are found in the `java.util.function` package and can be reused without having to define a custom interface. Standard functional interfaces are also used as parameters in the methods of the Java 8 Stream API.

Interface name	Abstract method	Description
Predicate<T>	boolean test(T t)	Represents a boolean-valued function of one argument.
Function<T, R>	R apply(T t)	Represents a function that accepts one argument and produces a result.
Consumer<T>	void accept(T t)	Represents an operation that accepts a single input argument and returns no result.
Supplier<T>	T get()	Represents a supplier of results.
UnaryOperator<T>	T apply(T t)	Represents an operation on a single operand that produces a result of the same type as its operand.
BinaryOperator<T>	T apply(T t1, T t2)	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

Table 1: Standard functional interfaces from the `java.util.function package` as described in [1]

Listing 5 shows an exapmle using 4 Functional Interfaces from the `java.util.function` package. The example first uses a Supplier to generate a list of integers. Then it uses a Consumer which is passed to the `forEach` method, thus execting the consumer lambda body for each element in the list. In the Consumer body, we use a Predicate to test if the number is even. If yes, then we use a Function to compute its square and print it to the standard output.

```
Supplier<List<Integer>> numberSupplier = () -> Arrays.asList(1, 2, 3, 4);

Predicate<Integer> isEven = x -> x % 2 == 0;

Function<Integer, Integer> getSquare = x -> x * x;

Consumer<Integer> ifEvenPrintSquare = x -> {
    if (isEven.test(x)) {
        System.out.println(getSquare.apply(x));
    }
};

List<Integer> numbers = numberSupplier.get();
numbers.forEach(ifEvenPrintSquare);
```

```
OUTPUT-----------------------------------------------------------------------
4
16
```

Listing 5: Example of usage for standard functional interfaces

The most popular way in which lambda expressions and standard functional interfaces are used is passing them as parameters to methods from the Java 8 APIs:

- The *Stream API*:

  `filter(Predicate<? super T>)`

  `map(Function<? super T,? extends R>)`

  `reduce(BinaryOperator<T>)`

- The *Collections API*:

  `forEach(Consumer<? super T>)`

  `removeIf(Predicate<? super E>)`

- The *Optional* class:

  `ifPresent(Consumer<? super T>)`

  `orElseGet(Supplier<? extends T>)`

7 Conclusion

Lambda expressions were introduced in Java 8, adding functional programming features to the existing object-oriented language. It provided a more elegant and consise way to create instances of functional interfaces as compared to anonymous classes, with an additional benefit of increased performance.

A lambda expression represents an unnamed block of code whose type is inferred by the compiler based on the context it is used in. A lambda expression cannot be used on its own, but can be used in assignments, returns or as a method parameter. This feature enables us to pass blocks of code to methods creating behavior parametrization and avoiding duplicate code.

A number of standard functional interfaces were created and are widely used in the Java 8 Stream and Collections APIs. This provides new and improved ways of processing data.

References

[1] Java Platform SE 8 Oracle Documentation. .

[2] Alan Mycroft by Raoul-Gabriel Urma, Mario Fusco. *Java 8 in Action: Lambdas, Streams, and functional-style programming.* Manning Publications, 2014.

[3] Davood Mazinanian, Ameya Ketkar, Nikolas Tsantalis, and Danny Dig. Understanding the Use of Lambda Expressions in Java. 1, 10 2017.

[4] Kishori Sharan. *Beginning Java 8 Language Features: Lambda Expressions, Inner Classes, Threads, I/O, Collections, and Streams.* Apress, 1 edition, 2014.

[5] Venkat Subramaniam. *Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions.* Pragmatic Bookshelf, 1 edition, 2014.

[6] Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming.* O'Reilly Media, 2014.