How to start working with Lambda Expressions in Java

Luis Santiago Follow

Dec 23, 2017 · 3 min read

Before Lambda expressions support was added by JDK 8, I'd only used examples of them in languages like C# and C++.

Once this feature was added to Java, I started looking into them a bit closer.

The addition of lambda expressions adds syntax elements that increase the expressive power of Java. In this article, I want to focus on foundational concepts you need to get familiar with so you can start adding lambda expressions to your code today.

Quick Introduction

Lambda expressions take advantage of parallel process capabilities of multi-core environments as seen with the support of pipeline operations on data in the Stream API.

They are anonymous methods (methods without names) used to implement a method defined by a functional interface. It's important to know what a functional interface is before getting your hands dirty with lambda expressions.

Functional interface

A functional interface is an interface that contains one and only one abstract method.

If you take a look at the definition of the Java standard Runnable interface, you will notice how it falls into the definition of functional interface because it only defines one method: `run()`.

In the code sample below, the method `computeName` is implicitly abstract and is the only method defined, making MyName a functional interface.

```
interface MyName{
  String computeName(String str);
}
```

Functional Interface

The Arrow Operator

Lambda expressions introduce the new arrow operator `->` into Java. It divides the lambda expressions in two parts:

```
(n) -> n*n
```

The left side specifies the parameters required by the expression, which could also be empty if no parameters are required.

The right side is the lambda body which specifies the actions of the lambda expression. It might be helpful to think about this operator as "becomes". For example, "n becomes n*n", or "n becomes n squared".

With functional interface and arrow operator concepts in mind, you can put together a simple lambda expression:

```
interface NumericTest {
        boolean computeTest(int n);
}

public static void main(String args[]) {
        NumericTest isEven = (n) -> (n % 2) == 0;
        NumericTest isNegative = (n) -> (n < 0);

        // Output: false
        System.out.println(isEven.computeTest(5));
```

Numeric Test

```
interface MyGreeting {
        String processName(String str);
}

public static void main(String args[]) {
        MyGreeting morningGreeting = (str) -> "Good Morning
        MyGreeting eveningGreeting = (str) -> "Good Evening

        // Output: Good Morning Luis!
        System.out.println(morningGreeting.processName("Lui
```

Greeting Lambda

The variables `morningGreeting` and `eveningGreeting`, lines 6 and 7 in the sample above, make a reference to `MyGreeting` interface and define different greeting expressions.

When writing a lambda expression, it is also possible to explicitly specify the type of the parameter in the expression like this:

```
MyGreeting morningGreeting = (String str) -> "Good Morning "
MyGreeting eveningGreeting = (String str) -> "Good Evening "
```

Lambda with Type

Block Lambda Expressions

So far, I have covered samples of single expression lambdas. There is another type of expression used when the code on the right side of the arrow operator contains more than one statement known as **block lambdas**:

```
interface MyString {
        String myStringFunction(String str);
}

public static void main (String args[]) {
        // Block lambda to reverse string
        MyString reverseStr = (str) -> {
                String result = "";

                for(int i = str.length()-1; i >= 0; i--)
                        result += str.charAt(i);

```

Block Lambda

Generic Functional Interfaces

A lambda expression cannot be generic. But the functional interface associated with a lambda expression can. It is possible to write one generic interface and handle different return types like this:

```
interface MyGeneric<T> {
        T compute(T t);
}

public static void main(String args[]){

        // String version of MyGenericInteface
        MyGeneric<String> reverse = (str) -> {
                String result = "";

                for(int i = str.length()-1; i >= 0; i--)
                        result += str.charAt(i);

                return result;
        };

        // Integer version of MyGeneric
        MyGeneric<Integer> factorial = (Integer n) -> {
                int result = 1;

                for(int i=1; i <= n; i++)
                        result = i * result;
```

Generic Functional Interface

Lambda Expressions as arguments

One common use of lambdas is to pass them as arguments.

They can be used in any piece of code that provides a target type. I find this exciting, as it lets me pass executable code as arguments to methods.

To pass lambda expressions as parameters, just make sure the functional interface type is compatible with the required parameter.

```
interface MyString {
        String myStringFunction(String str);
}

public static String reverseStr(MyString reverse, String st
  return reverse.myStringFunction(str);
}

public static void main (String args[]) {
        // Block lambda to reverse string
        MyString reverse = (str) -> {
                String result = "";

                for(int i = str.length()-1; i >= 0; i--)
                        result += str.charAt(i);
```

Lambda Expression as an Argument

These concepts will give you a good foundation to start working with lambda expressions. Take a look at your code and see where you can

increase the expressive power of Java.