ABOUT DANI BUIZA

Daniel Gutierrez Diez holds a Master in Computer Science Engineering from the University of Oviedo (Spain) and a Post Grade as Specialist in Foreign Trade from the UNED (Spain). Daniel has been working for different clients and companies in several Java projects as programmer, designer, trainer, consultant and technical lead.

Java 8 Lambda Expressions Tutorial

Posted by: Dani Buiza in Lambdas July 4th, 2014

In this article we are going to explain what Lambdas are, why are they important and how do they look like in Java. We are going to see also a couple of examples and real life applications.

All examples are being implemented using Eclipse Luna version 4.4 and Java version 8 update 5.

Lambdas?

A Lambda, in general, is a function that expects and accepts input parameters and produce output results (it may also produce some collateral changes).

Java offers Lambdas as one of its main new features since Java 8.

A Lambda is an instance of a functional interface (until Java 8, these were called single abstract method interfaces, SAM interfaces, like

```
Runnable
```

,

```
Comparator
```

or

```
Callable
```

) and with Java supporting them, it is possible to pass functions around like we are used to do with parameters and data in general.

Before, we used anonymous inner classes to instantiate objects of functional interfaces, with Lambdas, this can be simplified. It is possible to annotate these functional interfaces using the new annotation

```
@FunctionalInterface
```

.

How do they look like in Java?

In my opinion Lambdas are the largest enhancement done in Java since the Java 5 Generics from a language point of view.

Basically, they are composed of an arguments section that can be empty, containing 0, 1 or more input parameters, an arrow (->) and a body that can be almost any kind of Java piece of code.

These are valid examples in pseudo code:

```
(argument) -> { body } // just one input parameter
```

```
(argument1, argument2...) -> { body } // more than one input parameter
```

And in Java:

```
(x, y) -> x + y;
```

: in this example we have two input parameters and one output composed of the adition of both of them.

- In this one we have an input parameter and an action to be executed, in this case to print out in the console the string passed as input:

```
str -> { System.out.println(str); };
```

. We see that there are no output results.

- Another example is to print all elements of List:

```
Arrays.asList( "1", "2", "3" ).forEach( e -> System.out.println( e ) );
```

. In this case, we are using as input parameter every single element of the list and the action to be executed is to print them out in the console

console.

First examples

We are going to see some real examples. The code bellow creates a binary function that returns the addition of two input values:

```
/* functions with two input parametes and one output can be implemented easily using lambdas */
    BiFunction addition = ( x, y ) -> x + y;
    System.out.println( "calling addition of 2 and 3 resulting: " + addition.apply( 2, 3 ) );
```

we can use the function by calling its

```
apply()
```

method.

The next piece of code shows how to implement the

```
Runnable
```

interface without declaring its execute method:

```
/*
 * Runnable is a functional interface, we have to implement the method run() or provide a
 * function for it in Lambda style
 */
Runnable r = ( ) -> System.out.println( "run run run" );
Thread t = new Thread( r );
t.start();
```

This is possible using Lambdas since

```
Runnable
```

is a functional interface.

The following one prints all list entries in the console using Lambdas as well:

```
/* prints out in the console all elements of the list using the for each functionality */
Arrays.asList( "element", "other", "another one" ).forEach( e -> System.out.println( e ) );
```

We can see that we are using the

```
forEach
```

loop introduced as well in Java 8.

We can use a Lambda expression to implement a

```
Comparator
```

of Strings; this is possible because

```
Comparator
```

is a functional interface as well:

```
/* sorts all elements in a collection using a lambda expression as comparator */
List names = Arrays.asList( "prado", "gugenheim", "reina sofia", "louvre" );
Collections.sort( names, ( String a, String b ) -> b.compareTo( a ) );
names.forEach( e -> System.out.println( e ) );
```

At the end of this article you will find a link where you can download all the examples shown here and many more.

Real life example

Until here, we did see some small examples of how Lambdas look like in Java and for what they can be useful, now we are going to solve a real life problem by using them.

Imagine that we have a collection of persons and we want to produce and print out some statistics about which these persons are, where they come from and what do these persons like to do.

First, in "normal" Java we would iterate through the collection and filter and group them by favorite actions. Something like:

```
List persons = populatePersons();
    for(Person p: persons){
       showSpanish(p);
       showEnglishSpeakers(p);
       showPeopleThatLikeDancing(p);

    }
```

and as example the

```
showSpanish()
```

method would look like:

```
if( p != null && Country.SPAIN.equals( p.getCountry() ) )
    {
       System.out.println( p );
    }
```

and similar code for the methods

```
showEnglishSpeakers()
```

and

```
showPeopleThatLiveDancing()
```

.

Imagine that we want to show also the persons that come from USA and like sports, so we implement a new method showAmericanThatLikeSports():

```
if( p != null && Country.USA.equals( p.getCountry() ) && Hobbie.SPORTS.equals( p.getFavouriteAction() ) )
    {
        System.out.println( p );
    }
```

so at the end we will have a bunch of methods that look the same and actually are doing the same: printing the persons information if a filter applies. If we would need to store these people grouped by country, language or hobbie and process these groups separately to produce some results, we would need to iterate again through all the created groups. I think there is a better way to do this.

For example we can refactor this code and create a show method that receives a filter as parameter:

```
private static void showPerson(Person p, Filter filter){
    if( filter.filter(p) )
    {
        System.out.println( p );
    }
}
```

Well, this method looks really good but we still have the problem of grouping the persons by different parameters like language, age, name or country. Another problem now is how to pass this filter to the

`showPerson`

method.

We can create an interface with a filter method and we can implement this interface inline when calling the

`showPerson()`

. Something like:

```
showPerson( p, new Filter()
{
    @Override
    public boolean filter( Person p )
    {
        return ( p != null && Country.SPAIN.equals( p.getCountry() ) );
    }
} );
showPerson( p, new Filter()
{
    @Override
    public boolean filter( Person p )
    {
        return ( p != null && Country.USA.equals( p.getCountry() ) && Hobbie.SPORTS.equals( p
                .getFavouriteAction() ) );
    }
} );
showPerson( ...
```

There are some things I do not like here: we are using inline classes, which are not very nice, we are writing duplicated code again and we did not resolve the problem of grouping the persons and produce results from these groups without iterate through them again. For solving the problem with the inline classes and the duplication of code we can create a separate class for the filter and instantiating it each time with the proper filter, but this that not solve the problem of produce results based on groups of persons. Also we would end up implementing a Filter class that is not going to be very self explaining; this looks more like a hack.

So what can we do? Since we are iterating through a collection, filtering its elements applying different criteria and executing the same action (more or less) for all of them, we can use Lambdas as we saw before in this article.

First of all we are going to print all persons using Lambdas:

```
persons.forEach( ( s ) -> {System.out.println( s );} );
```

we do not need to iterate in the old fashion way any more, we just use a for each loop.

If we want to apply some filter we can make use of Predicates. Predicates are functional interfaces that contains a test() method and returns a boolean value. We declare them as:

```
Predicate spanish = p -> { return Country.SPAIN.equals( p.getCountry() );};
```

for filtering the Spanish ones out, and use them as follows:

```
persons.stream().filter(spanish).forEach( ( person ) -> {  System.out.println( person ); } );
```

or we can use directly predicates and consumers as Lambdas without need to declare and instantiate them:

```
persons.stream().filter(
    person -> {return Country.SPAIN.equals( person.getCountry() );}).
    forEach((eachPerson) -> {System.out.println(eachPerson);});
```

Here we are using the

`stream()`

method from the Stream API as well. And if we want to show different statistics like the ones that come from china and like to eat, we can write:

```
persons.stream().filter(
        per-> {return Country.CHINA.equals( per.getCountry() )
        && Hobbie.EAT.equals( per.getFavouriteAction() );}).
        forEach( ( eachPerson ) -> {System.out.println( eachPerson );} );
```

It is also possible to combine different predicates by chaining them using logical operators like

`or()`

,

`and()`

...

```
Predicate germans = person -> {
    return Country.GERMANY.equals( person.getCountry() );
};

Predicate dancers = person -> {
```

```
    return Hobbie.DANCE.equals( person.getFavouriteAction() );
};

// combining different predicates is also possible
persons.stream().filter( germans.and( dancers ) ).forEach( ( eachPerson ) -> {
    System.out.println( eachPerson );
} );
```

It is also possible to get statistics for different person groups in a very sofisticated and clean way using the Stream API in combination with Lambdas:

```
// get statistics directly like number of Persons based of different criteria
persons.stream().filter( germans.and( dancers ) ).count();

// oldest person
System.out.println( "oldest one: "
    + persons.stream().filter( germans.and( dancers ) ).max( ( p1, p2 ) -> p1.getAge() - p2.getAge() ) );
```

Now is also easy to produce results based on groups of persons without need to store them separately and handle these groups afterwards.This is really powerful and the code is very easy to change and adapt to new requirements. It is also more legible than creating inline or helper classes.

So a final application would look like:

```
List persons = populatePersons();

// print all persons using lambdas
persons.forEach( ( s ) -> {
    System.out.println( s );
} );

// using an instantiated predicate
Predicate spanish = p -> {
    return Country.SPAIN.equals( p.getCountry() );
};
persons.stream().filter( spanish ).forEach( ( person ) -> {
    System.out.println( person );
} );

// using Lambdas directly for filtering the spanish ones
persons.stream().filter( person -> {
    return Country.SPAIN.equals( person.getCountry() );
} ).forEach( ( eachPerson ) -> {
    System.out.println( eachPerson );
} );

// using Lambdas directly for filtering the chinesse ones that love to eat
persons.stream().filter( person -> {
    return Country.CHINA.equals( person.getCountry() ) && Hobbie.EAT.equals( person.getFavouriteAction()
);
} ).forEach( ( eachPerson ) -> {
    System.out.println( eachPerson );
} );

// using Lambdas directly for filtering the chinesse ones that love to eat
persons.stream().filter( person -> {
    return Country.CHINA.equals( person.getCountry() ) && Hobbie.EAT.equals( person.getFavouriteAction()
);
} ).forEach( ( eachPerson ) -> {
    System.out.println( eachPerson );
} );

Predicate germans = person -> {
    return Country.GERMANY.equals( person.getCountry() );
};

Predicate dancers = person -> {
    return Hobbie.DANCE.equals( person.getFavouriteAction() );
};

// combining different predicates is also possible
persons.stream().filter( germans.and( dancers ) ).forEach( ( eachPerson ) -> {
    System.out.println( eachPerson );
} );

// using streams and filters is possible to get statistics directly like number of Persons
// based of different criteria
persons.stream().filter( germans.and( dancers ) ).count();

// oldest person
System.out.println( "oldest one: "
    + persons.stream().filter( germans.and( dancers ) ).max( ( p1, p2 ) -> p1.getAge() - p2.getAge() )
);
```

This looks much better. The only possible issue that you would find here, is that before, without lambdas, we could print out all kind of statistics by iterating just once through the collection (by duplicating code and using inline classes), and now, with Lambdas, we have to iterate several times (or use several times the collection stream and the forEach loop) in order to produce the similar results. The code looks much better, is easier to test, to change and to maintain, but the performance is worse. On the other hand we can produce statistics and results based on groups inside the collection without need to store them separately and iterate again and again, what can affect also the performance in a bad way. So we can see that we have many benefits but also some disadvantages.

In case that this happens and several iterations are worse than several computations from a performance point of view and we do not need to produce results based on groups of entries in the collection, we should go for a mixed solution by using Lambdas in a single iteration, or, if possible, to use the new Java 8 features that support parallelism.

We saw that the Lambda expressions and statements are very useful in combination with the new Stream and the Collections API. For more information about the streams API and other features coming with the Java 8 release, please visit http://www.javacodegeeks.com/2014/05/java-8-features-tutorial.html.

That is all. In this article we saw several Lambdas examples and the application of Lambdas in real life problems. We mentioned as well the Stream APi and we made some examples of its usage, we also saw that there are scenarios where it may better to combine Lambdas with "old" Java features and do not abuse of the stream API.

The java.util.function package

As reference we are going to list the main classes, annotations and interfaces introduced in the java 8 release that are relevant to the Lambda expressions and statements:

```
@FunctionalInterface
```

: Annotates an interface with just one abstract method.

```
Functions
```

: Functional interface that has an apply() method.

```
Predicates
```

: Functional interface that has an test() method. Useful for filtering mechanisms.

```
Suppliers
```

: Contains get methods. They work as supplier of objects (for example in factories).

```
Consumers
```

: Contains an accept() method. It is expected to produce side effects since it returns no results.

For all the interfaces mentioned above there are specific typified ones like

```
LongUnaryOperator
```

,

```
ObjIntConsumer
```

,

```
ToLongFunction
```

and many others. In the official oracle page you can find more information about this package and its classes, annotations and interfaces http://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html.

You can find all code used in this article in the following link: lambdas