## Source code related to Java 8

Compatible eclipse need to download, set java home first for maven, setting path at cmd only results mvn build error . Below is maven build.



Processing Collections with Streams API

In the previous article “[Traversing, filtering, processing collection, & methods enhancements with Lambda](http://www.javacodegeeks.com/2014/12/java-se-8-new-features-tour-traversing-filtering-processing-collection-methods-enhancements-with-lambda.html)”; I have deeply dived into explanation, and exploration on how to **traverse the collections** using lambda expression and with **method references**, filtering them with *predicate* interface, implementing **default** methods in interfaces, and finally implementing **static methods** in interfaces.

* Source code is hosted on my **Github** account: clone it from [here](https://github.com/mohamed-taman/JavaSE8-Features).

Table of Contents

1. Traversing collections with streams.

2. Creating streams from collections and arrays.

3. Aggregating stream values.

## 1. Traversing collections with streams

* A collection based stream isn’t like an input or an output stream.

Instead of dealing with each item individually, when you use streams, you don’t have to worry about the details of looping or traversing. You create a stream object directly from a collection. And then you can do all sorts of things with it including traversing, filtering, and aggregating its values. I’ll start with this example in the package eg.com.tm.java8.features.stream.traversing of project Java8Features. In a class code SequentialStream, in Java SE 8 there are two kinds of collection streams known as sequential and parallel streams.

List<person> people = new ArrayList<>();

people.add(new Person("Mohamed", 69));

people.add(new Person("Doaa", 25));

people.add(new Person("Malik", 6));

Predicate<person> pred = (p) -> p.getAge() > 65;

displayPeople(people, pred);

private static void displayPeople(List<person> people, Predicate<person> pred) {

* 14
* 15 System.out.println("Selected:");
* 16 people.forEach(p -> {
* 17 if (pred.test(p)) {
* 18 System.out.println(p.getName());
* 19 }
* 20 });
* 21 }
* A sequential stream is the simpler of the two and just like an iterator it will let you deal with each item in a collection one at a time. But with less syntax than before. In this code, I’ve created an array list of people, cast as a list. And it has three instances of a complex object, a class named Person. Then I’m using a Predicate to declare a condition, and displaying people that only satisfy the condition. From lines 48 to 52 in the displayPeople() method, I’m traversing the collection, looping through the data, and testing each item one at a time. run the code and you should have the following results:
* 1 Selected:
* 2 Mohamed

I’ll show you how to re-factor this code using a stream object. First, I’m going to comment out these lines of code. Now, below the commented code, I’ll start with the collection object. People. And then I’ll call a new method called stream. A stream object, just like the collection itself, has a generic declaration. If you get a stream from a collection, the items in the stream are of the same type as the collection itself. My collection has instances of the person class so the stream uses the same generic type.

* 1 System.out.println("Selected:");
* 2 // people.forEach(p -> {
* 3 // if (pred.test(p)) {
* 4 // System.out.println(p.getName());
* 5 // }
* 6 });
* 7
* 8 people.stream().forEach(p -> System.out.println(p.getName()));
* 9 }
* You call stream as a method, and now you have a stream object that you can do things with. I’ll start with a simple call to the four each method, and this will require a Lamda expression. I’ll pass in the argument. That’s the item in the list that I’m dealing with on this time through the iteration. Then the Lambda operator and then the implementation of the method. And I’ll use simple system output and I’ll output the person’s name. I’ll save and displaying all of the people run the code and there’s the result. Because I’m not filtering anymore, I’m in the list.
* 1 Selected:
* 2 Mohamed
* 3 Doaa
* 4 Malik
* Now, once you have a stream this is how easy it’ll be to use a predicate object. When I use the for each method and dealt with each item one at a time. I had to explicitly call the test method of the predicate. But using a stream you can call a method named filter. That expects a predicate object, and all predicates have a test method, and so it already knows how to call that method. So, I’ll break up this code a bit. I’ll move the call to the .forEach() method down a couple of lines, and then on the empty line in the middle, I’ll call the new filter method.
* 1 people.stream()
* 2 .filter(pred)
* 3 .forEach(p -> System.out.println(p.getName()));
* The filter method expects an instance of the predicate interface. And I’ll pass my predicate object in. The filter method returns the stream but now the filtered version, and from there I can call the forEach() method. I’ll run the code and now I am only displaying items from the collection that satisfy the predicate condition. You can do lot more with the streams. Take a look a the documentation for streams in the Java SE 8 API docs.
* 1 Selected:
* 2 Mohamed
* And you’ll see that in addition to filtering you can also aggregate and do all sorts of other things with streams. Before I conclude this demonstration though, I want to show you a very important distinction between sequential and parallel streams. One of the goals of the stream API in Java SE 8 is to let you break up processing on a system that has multiple CPUs. This multi CPU processing is handled automatically by the Java runtime. All you need to do is turn your sequential stream into a parallel stream.
* And there are a couple of ways of doing that syntactically. I’ll make a copy of my sequential stream class. I’ll go to my package explorer, and I’ll copy it and paste it. And I’ll name the new class, ParallelStream. And I’ll open the new class. In this version, I’ll get rid of the commented code. I don’t need that anymore. And now here are two ways of creating a parallel stream. One approach is to call a different method from the collection. Instead of stream I’ll call parallelStream(). And now I have a stream that will automatically be broken down and allocated to different processors.
* 1 private static void displayPeople(List<person> people, Predicate<person> pred) {
* 2 System.out.println("Selected:");
* 3 people.parallelStream()
* 4 .filter(pred)
* 5 .forEach(p -> System.out.println(p.getName()));
* 6 }
* I’ll run the code and I’ll see that it’s doing exactly the same thing, filtering and returning the data.
* 1 Selected:
* 2 Mohamed
* Here is the other way to create a parallel stream. I’ll call this stream() method again. And then from the stream method I’ll call a method named parallel() and that does exactly the same thing. I start with the sequential stream and I end up with a parallel stream. It’s still a stream. It can still filter, it can still process in exactly the same way as before. But now it’ll be broken up where possible.
* 1 people.stream()
* 2 .parallel()
* 3 .filter(pred)
* 4 .forEach(p -> System.out.println(p.getName()));

### Conclusion

* There isn’t any clear prescription for when to use a parallel stream over a sequential stream. It depends on the size and complexity of your data, and the capabilities of the hardware. The multi CPU system that you’re running on. The only recommendation I can give you is to try it with your application and your data. Set up benchmarks, timing the operation. Use a sequential stream and use a parallel stream and see which works better for you.
* 2. Creating streams from collections and arrays
* Introduction
* you can also create streams directly from arrays.
* How it works
* In this project Java8Features, in the package eg.com.tm.java8.features.stream.creating, I have a class namedArrayToStream. And in its main method, I’ve created an array of three items. And they’re each instances of my complex object, the Person class.
* 01 public static void main(String args[]) {
* 02
* 03 Person[] people = {
* 04 new Person("Mohamed", 69),
* 05 new Person("Doaa", 25),
* 06 new Person("Malik", 6)};
* 07 for (int i = 0; i < people.length; i++) {
* 08 System.out.println(people[i].getInfo());
* 09 }
* 10 }
* This class has setters and getters for the private fields, and the new getInfo() method, to return a concatenated string.
* 1 public String getInfo() {
* 2 return name + " (" + age + ")";
* 3 }
* Now if you wanted to use a stream to process this array, you might think you would need to convert it to an array list, perhaps and then from there create the stream. But it turns out there are a couple of ways to go directly from an array to a stream. Here’s the first approach. I’m not going to need these three lines of code that I’m using to process the data. So I’ll comment those out. And then down here, I’ll declare an object for the type is stream.
* Stream is an interface, which is a member of java.util.stream. When I press Ctrl+Space and select it from the list, I’m asked for the generic type of the items, that the stream will manage. And these will be items of type Person, just like the items in the array itself. I’ll name my new stream object, stream, in all lower case. And here’s the first way to create the stream. Use the stream interface again, and call a method named of(). Notice that there are a couple of different versions.
* One that takes a single object, and one that takes a series of objects. I’ll use the one that takes one argument, and I’ll pass in my array, people, and that’s all I need to do. Stream.of() means take this array and wrap it inside a stream. And now i can use lambda expressions, filters, method references and other things that work on Stream objects. I’ll call the stream objects for each method and i will pass in a lambda expression, i’ll pass in the current person and then after the lambda operator, i’ll output the person’s information. Using the object’s getInfo() method.
* 01 Person[] people = {
* 02 new Person("Mohamed", 69),
* 03 new Person("Doaa", 25),
* 04 new Person("Malik", 6)};
* 05
* 06 // for (int i = 0; i < people.length; i++) {
* 07 // System.out.println(people[i].getInfo());
* 08 // }
* 09 Stream<Person> stream = Stream.of(people);
* 10 stream.forEach(p -> System.out.println(p.getInfo()));
* I’ll save and run the code and there’s the result. I’m outputting the items in the same order in which they were placed in the array. So, that’s one approach using Stream.of().
* 1 Mohamed (69)
* 2 Doaa (25)
* 3 Malik (6)
* There’s another approach that does exactly the same thing. I’m going to duplicate that line of code, and comment out one version. And this time in using Stream.of(), I’ll use a class named Arrays, which is a member of the package java.util.
* And from there, I’ll call a method named stream. Notice the stream method can be wrapped around arrays of a variety of types. Including both primitives and complex objects.
* 1 // Stream<person> stream = Stream.of(people);
* 2
* 3 Stream<person> stream = Arrays.stream(people);
* 4 stream.forEach(p -> System.out.println(p.getInfo()));
* I’ll save and run that version and the stream does exactly the same thing as before.
* 1 Mohamed (69)
* 2 Doaa (25)
* 3 Malik (6)
* Conclusion
* So either Stream.of() or Arrays.stream() will do exactly the same thing. Take an array of primitive values or complex objects and turn them into a stream, that you can then use with lambdas, filters, and method references.

## 3. Aggregating stream values

* Introduction
* I’ve previously described how to use a stream to iterate over a collection. But you can also use streams to aggregate items in a collection. That is, calculate sums, averages, counts, and so on. When you do this kind of operation, it’s important to understand the nature of parallel streams.
* How it works
* So I’m going to start this demonstration in the project Java8Features, in the packageeg.com.tm.java8.features.stream.aggregating. And I am going to first work with the class ParallelStreams. In this class’smain method I’ve created an array list containing string items.
* I’m using a simple for loop I’ve added 10,000 items to the list. Then on lines 35 and 36, I’m creating a stream and using the for each method and outputting each stream one at a time.
* 01 public static void main(String args[]) {
* 02
* 03 System.out.println("Creating list");
* 04 List<string> strings = new ArrayList<>();
* 05 for (int i = 0; i < 10000; i++) {
* 06 strings.add("Item " + i);
* 07 }
* 08 strings.stream()
* 09 .forEach(str -> System.out.println(str));
* 10 }
* When I run this code, I get an expected result. The items are output to the screen in the same order in which they were added to the list.
* 01 .........
* 02 Item 9982
* 03 Item 9983
* 04 Item 9984
* 05 Item 9985
* 06 Item 9986
* 07 Item 9987
* 08 Item 9988
* 09 Item 9989
* 10 Item 9990
* 11 Item 9991
* 12 Item 9992
* 13 Item 9993
* 14 Item 9994
* 15 Item 9995
* 16 Item 9996
* 17 Item 9997
* 18 Item 9998
* 19 Item 9999
* Now let’s see what happens when we turn this into a parallel stream. As I described previously, I can do this either by calling the parallel stream method or by taking the results of stream and passing those to parallel.
* I’ll do the latter. Now I’m working with a parallel stream, that is a stream that can be broken up and the work load split among multiple processors.
* 1 strings.stream()
* 2 .parallel()
* 3 .forEach(str -> System.out.println(str));
* I’ll run the code again and watch what happens, notice that the last item printed is not the last item in the list. That would’ve been 9,999. And if I scroll around in the output I’ll see that the processing is jumping around in someway. What’s happening is that the run time is arbitrarily splitting the data into blocks.
* 01 .........
* 02 Item 5292
* 03 Item 5293
* 04 Item 5294
* 05 Item 5295
* 06 Item 5296
* 07 Item 5297
* 08 Item 5298
* 09 Item 5299
* 10 Item 5300
* 11 Item 5301
* 12 Item 5302
* 13 Item 5303
* 14 Item 5304
* 15 Item 5305
* 16 Item 5306
* 17 Item 5307
* 18 Item 5308
* 19 Item 5309
* 20 Item 5310
* 21 Item 5311
* And then handing each block to an available processor. It’s only after all of the blocks have been processed that my next bit of Java code would be executed. But internally, within the call to the forEach() method, all of this work is being split up as needed. Now this might or might not provide a performance benefit. It depends on the size of your data set. And the nature of your hardware. But one of the things that this example shows you is that if you need to process items sequentially, that is one at a time in the same order in which they were added to the collection, then a parallel stream might not be the way to do it.
* Sequential streams can guarantee they’re working in the same order every time. But a parallel stream, by definition, is going to do things in the most efficient way possible. So parallel streams are especially useful when you’re aggregate operations. Where you’re taking into account all of the items in a collection and then creating a some sort of aggregate value from them. I’ll show you examples of counting items in a collection, averaging them, and summing them using strings.
* In this class, CountItems in the main method, I’m starting with the same basic code. Creating 10,000 strings in a list. And then there’s a for each method that’s looping through and handling them one at a time.
* 01 public static void main(String args[]) {
* 02
* 03 System.out.println("Creating list");
* 04 List<string> strings = new ArrayList<>();
* 05 for (int i = 0; i < 10000; i++) {
* 06 strings.add("Item " + i);
* 07 }
* 08 strings.stream()
* 09 .forEach(str -> System.out.println(str));
* 10 }
* In this example, instead of processing each string individually, I instead want to count them up. So, I’ll comment out that code, and here’s the code I’ll use. Since I don’t know exactly how many items are going to be in the collection. I’ll cash the result I am about to create as a long integer.
* And I’ll name it count, and I’ll get it’s values by calling strings. That’s my collection, .stream(), .count(), and this returns a long value. Then I’ll use system output and I’ll report the result. With count: and then I’ll append my result.
* 1 // strings.stream()
* 2 // .forEach(str -> System.out.println(str));
* 3 long count = strings.stream().count();
* 4 System.out.println("Count: " + count);
* I’ll save my changes and run the code and there’s the result. The count of the items in the collection is almost instantaneous.
* 1 Creating list
* 2 Count: 10000
* Now to make this a little bit more dramatic I’ll add a couple of zeros here and now I’m dealing with 1,000,000,000 strings. I’ll run the code again and the result comes back again almost instantly.
* 1 Creating list
* 2 Count: 1000000
* Now watch what happens if I instead parallelize the string. I’ll add dot parallel here:
* 1 // strings.stream()
* 2 // .forEach(str -> System.out.println(str));
* 3 long count = strings.stream().parallel().count();
* 4 System.out.println("Count: " + count);
* And then I’ll run the code, and it takes a little bit longer. Now, I could benchmark how long it’s taking these operations to happen, by capturing the current time stamp before and after the operation. And then doing a little math. And what it would show might differ from one system to another. But in my experience when dealing with these sorts of simple collections containing simple values there isn’t much benefit to parallel streams. Your mileage may very though. And I encourage you to do your own benchmarking. But that’s how you would do a count.
* Let’s take a look at summing and averaging. I’ll go to my class SumAndAverage. This time, I have a list of three person objects, each with a different age. And my goal is to get the sum of the three ages, and the average of the three ages. I’ll add a new line of code after all the instances of the person class have been added to the list. And I’ll create an integer variable that I’ll name sum.
* I’ll start by getting a stream, using people.stream(). From there I’ll call a method called mapToInt(). Notice that there’s a Map Method. mapToDouble() and mapToLong() as well. The purpose of these methods is to take complex object and extract a simple primitive value from it, and create stream of those values, and you do this with Lambda expression. So, I’ll choosemapToInt() because the ages of each person are integers.
* For the Lambda expression, I’ll start with a variable that will represent the current person. Then the Lambda operator and then an expression that returns an integer. I’ll use p.getAge(). This returns something called an int string or a string of integers. There’s also a double string class and a few others. Now from this stream because I already knew it was a numeric value I can call a method named sum(). And that’s it. I’ve now summed up all the aged values from all the personals object in my collection. With a single statement, I’ll output the result using System Output. My label will be total of ages and I’ll append to that my sum.
* 1 List<person> people = new ArrayList<>();
* 2 people.add(new Person("Mohamed", 69));
* 3 people.add(new Person("Doaa", 25));
* 4 people.add(new Person("Malik", 6));
* 5
* 6 int sum = people.stream()
* 7 .mapToInt(p -> p.getAge())
* 8 .sum();
* 9 System.out.println("Total of ages " + sum);
* I’ll save my code and run it. And the total of all three ages is 100.
* 1 Total of ages 100
* Averaging these values is very similar. But because whenever you’re doing averaging your doing divisions, and you might get a divide by zero problem, and so, when you do an average, you’ll get back something called an Optional variable.
* And there are a number of types you can use for this. For my averaging, I’m going to expect a double value to come back. So, I’m going to create a variable, called OptionalDouble. Notice that there’s also Optional Int and Optional Log. I’ll name my variable Avg, for average. And I’ll use this same sort of code I just used to get the sum, starting with people.stream(). And then from there, I’ll use mapToInt() again. And I’ll pass in the same lambda expression that i use last time, and then from there I’ll call the average method.
* Now with an OptionalDouble object, before you process it you should always make sure that it actually has a double valueand you do this with a method named isPresent(). So, I’ll start off with an if else code template. And I’ll set my condition toavg.isPresent(). If that condition is true, I’ll use System Output. And I’ll label this just Average. And I’ll append my average variable. In the else clause I’ll simply say that the average wasn’t calculated.
* 1 OptionalDouble avg = people.stream()
* 2 .mapToInt(p -> p.getAge())
* 3 .average();
* 4 if (avg.isPresent()) {
* 5 System.out.println("Average: " + avg);
* 6 } else {
* 7 System.out.println("average wasn't calculated");
* 8 }
* Now in this example I know it will be successful because I’ve provided ages to all three people but that won’t always be the case. Like I said if you end up with a divide by zero situation you might not get a double value back. I’ll save and run the code, and notice that with the optional double class, it’s a complex object.

1 Total of ages 100

2 Average: OptionalDouble[33.333333333333336]

So the type is wrapped around the actual value. I’ll go to this code, where I’m referencing the object directly, and I’ll call it’sgetAsDouble() method.

if (avg.isPresent()) {

System.out.println("Average: " + avg.getAsDouble());

} else {

System.out.println("average wasn't calculated");

}

And now I’ll get back a primitive double value. I’ll run the code again and now the result

is what I was looking for.

1 Total of ages 100

2 Average: 33.333333333333336

Conclusion

So using streams and lambda expressions, you can easily calculate aggregate values from collections with a tiny, tiny amount of code.