JAVA 8

Java 8 Streams API Tutorial with Examples

October 9, 2015

Java 8 Streams API tutorial starts off with defining Java 8 Streams, followed by an explanation of the important terms making up the Streams definition. We will then look at Java 8 code examples showing how to exactly use Streams API. **By the end of this tutorial you should feel confident of writing your first program utilising Java 8 Streams API.**

What are Streams/Defining Streams

A Stream in Java is a sequence of elements supporting parallel and aggregate operations. This sequence of elements are obtained from a source. Further, streams in Java 8 support processing of these elements through operations defined in a declarative way. These operations are linked-up according to the principle of pipelines and they access the elements via Internal Iterations.

Conceptual terms making up the Streams definition

Streams definition given above has a lot of terms put together. To understand this definition in its totality we need to understand each of these terms. Lets look at them one-by-one:

- **Sequence of elements** – A stream provides an interface to a sequenced set of values of a specific type. For e.g. `Stream<Integer>` is a stream of type `Integer`.
- **Source of stream elements** – Streams are defined as originating from a specific source which can be collections, arrays, input-output(I/O) resources etc
- **Operations on stream elements** – As the stream's elements are encountered, several pre-defined operations can be declared to act on the stream elements to map, reduce and collect these elements.
- **Parallel and aggregate operations** – The operations working on these stream of elements can work in parallel on multi-core architectures. Aggregate operations act on elements in the stream in a sequence and end up aggregating data into an end value.
- **Pipeline of Operations** – Various operations which have been declared to act on a stream work together based on the concept of Pipelines(link to tutorial). I.e. output of one stream operation acts as input of the next stream operation.
- **Internal iterations**– Internal iterations delegate the work of iterating to the Streams library. The programmer just needs to specify in a declarative manner as to which operation has to be applied to the stream of elements.

Streams API usage example

To start with, let us look at a class `Employee.java` which has –

- 2 instance attributes – **name** & **age**.
- Getters and setters for these attributes.
- A constructor with both attributes.
- The `toString()` method.

Employee.java

```
package com.javabrahman.java8;
public class Employee{
  private String name;
  private Integer age;
  public Employee(String name, Integer age){
    this.name=name;
    this.age=age;
  }
  public String getName(){
    return name;
  }
  public void setName(String name){
    this.name=name;
  }
  public Integer getAge(){
    return this.age;
  }
  public void setAge(Integer age){
    this.age=age;
  }
  public String toString(){
    return "Employee Name:"+this.name
      +"  Age:"+this.age;
  }
}
```

Next I have a class called `BasicStreams.java`, given below, in which I have –

- A static list of employees called `employeeList`
- `main()` method where I have my streams-based filter logic
- I initialise my list of Employees with 5 Employee Objects using the 2-parameter `Employee` constructor which utilises the `Arrays.asList()` method

Streams usage example - BasicStreams.java

```
package com.javabrahman.java8;
import java.util.List;
import java.util.Arrays;
import static java.util.stream.Collectors.toList;
public class BasicStreams {
 static List<Employee> employeeList=
      Arrays.asList(new Employee("Tom Jones", 45),
                    new Employee("Harry Major", 25),
                    new Employee("Ethan Hardy", 65),
                    new Employee("Nancy Smith", 15),
                    new Employee("Deborah Sprightly", 29));
 public static void main(String args[]){
   List<Employee> filteredList = employeeList.stream()
                        .limit(2)
                        .collect(toList());
   filteredList.forEach(System.out::println);
 }
}
```

OUTPUT of the above code

```
Employee Name:Tom Jones Age:45
Employee Name:Harry Major Age:25
```

Explanation of the code

- The program starts with static import of `Collector` Class' static `toList()` method. This method is used to get a list from a stream
- In the `main()` method, **`stream()` is invoked on employeeList**. The method `stream()` has been newly introduced in Java 8 on the interface Collection which List interface extends. This `stream()` method gives an instance `java.util.Stream` as the return type
- The **stream of employees is then passed(or pipe-lined) to the function `limit()`**. The `limit()` function puts a limit to the maximum number of elements which will be picked from the stream. In the given example I have passed the value 2, hence the current stream is now limited to first 2 elements. Also, note that `limit()` is an intermediate operation, i.e. the stream processing does not end with `limit()` method.
- The `collect()` method is then invoked on the stream(which now has only 2 elements). `Collect()` uses a `Collector` of a specific type which in the given example is of type `List`(returned by the static `toList()` method of `Collectors` class). To simplify the previous statements, **`collect()` uses the `toList()` method to return a list equivalent of the stream** pipe-lined into it(named `filteredList`). Note that `collect()` is a terminal operation, i.e. the processing of the stream ends with the `collect()` method.
- At the end, I simply use a Java 8 style `forEach` loop and a Method Reference to the `System.out.println()` method to print all the elements in the resultant `filteredList`.
- As expected the 2 employee objects are printed using the overridden `Employee.toString()` method.

Summary

In the above tutorial we saw what Java 8 Streams are, understood the various terms which describe a Stream, and finally saw a basic example of how to start using Streams in your programs using the intermediate & terminal operations.

Java 8 Streams API tutorials on JavaBrahman

- ↪ Streams API – Introduction & Basics
- ↪ Understanding Stream Operations | Intermediate and Terminal Operations
- ↪ Mapping with Streams using map and flatMap methods
- ↪ Filtering and Slicing with Streams using filter,distinct,limit,skip methods
- ↪ Matching with Streams using allMatch,anyMatch,noneMatch methods
- ↪ Stream API's findFirst,findAny methods' tutorial
- ↪ 'Peeking' into a running Stream with Stream.peek() method
- ↪ Creating Infinite Streams with iterate\generate methods
- ↪ Reducing Streams using Streams.reduce method

Click on a category to view all articles

Algorithms & DS in Java (7)
Core Java (24)
Design Patterns (17)
Eclipse Plugins (3)
Error Handling (6)
General Java Programs (9)
Java 8 (60)
Java 9 & beyond... (4)
JavaScript and HTML (2)
JPA (6)
Latest in Tech (4)
Maven (1)
New Features in Java 7 (3)
Node.js (1)
Programming & Design Principles (9)
Quick Coding Tips (19)
Reviews (1)