A Guide to Java Streams

The addition of the *Stream* was one of the major features added to Java 8. This in-depth tutorial is an introduction to the many functionalities supported by streams, with a focus on simple, practical examples.

To understand this material, you need to have a basic, working knowledge of Java 8 (lambda expressions, *Optional,* method references).

Introduction

First of all, Java 8 Streams should not be confused with Java I/O streams (ex: *FileInputStream* etc); these have very little to do with each other.

Simply put, streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.
This functionality – *java.util.stream* – supports functional-style operations on streams of elements, such as map-reduce transformations on collections.

Let's now dive into few simple examples of stream creation and usage – before getting into terminology and core concepts.

Java Stream Creation

Let's first obtain a stream from an existing array:

```
private static Employee[] arrayOfEmps = {
    new Employee(1, "Jeff Bezos", 100000.0),
    new Employee(2, "Bill Gates", 200000.0),
    new Employee(3, "Mark Zuckerberg", 300000.0)};
Stream.of(arrayOfEmps);
```

We can also obtain a stream from an existing *list*:

```
private static List<Employee> empList = Arrays.asList(arrayOfEmps);
empList.stream();
```

Note that **Java 8 added a new *stream()* method to the *Collection* interface.**

And we can create a stream from individual objects using *Stream.of()*:

```
Stream.of(arrayOfEmps[0], arrayOfEmps[1], arrayOfEmps[2]);
```

Or simply using *Stream.builder()*:

```
Stream.Builder<Employee> empStreamBuilder = Stream.builder();
empStreamBuilder.accept(arrayOfEmps[0]);
empStreamBuilder.accept(arrayOfEmps[1]);
empStreamBuilder.accept(arrayOfEmps[2]);
Stream<Employee> empStream = empStreamBuilder.build();
```

There are also other ways to obtain a stream, some of which we will see in sections below.

Java Stream Operations

Let's now see some common usages and operations we can perform on and with the help of the stream support in the language.

forEach

forEach() is simplest and most common operation; it loops over the stream elements, calling the supplied function on each element.

The method is so common that is has been introduced directly in *Iterable, Map* etc:

```
@Testpublic void whenIncrementSalaryForEachEmployee_thenApplyNewSalary() {
    empList.stream().forEach(e -> e.salaryIncrement(10.0));


    assertThat(empList, contains(
      hasProperty("salary", equalTo(110000.0)),
      hasProperty("salary", equalTo(220000.0)),
      hasProperty("salary", equalTo(330000.0))
    ));}
```

This will effectively call the *salaryIncrement()* on each element in the *empList*.

***forEach()* is a terminal operation**, which means that, after the operation is performed, the stream pipeline is considered consumed, and can no longer be used. We'll talk more about terminal operations in the next section.

map

map() produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

The following example converts the stream of *Integer*s into the stream of *Employees*:

```
@Testpublic void whenMapIdToEmployees_thenGetEmployeeStream() {

    Integer[] empIds = { 1, 2, 3 };


    List<Employee> employees = Stream.of(empIds)

      .map(employeeRepository::findById)

      .collect(Collectors.toList());


    assertEquals(employees.size(), empIds.length);}
```

Here, we obtain an *Integer* stream of employee ids from an array. Each *Integer* is passed to the function *employeeRepository::findById()* – which returns the corresponding *Employee* object; this effectively forms an *Employee* stream.

collect

We saw how *collect()* works in the previous example; its one of the common ways to get stuff out of the stream once we are done with all the processing:

```
@Testpublic void whenCollectStreamToList_thenGetList() {

    List<Employee> employees = empList.stream().collect(Collectors.toList
());



    assertEquals(empList, employees);}
```

collect() performs mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in the *Stream* instance.

The strategy for this operation is provided via the *Collector* interface implementation. In the example above, we used the *toList* collector to collect all *Stream* elements into a *List* instance.

filter

Next, let's have a look at *filter()*; this produces a new stream that contains elements of the original stream that pass a given test (specified by a Predicate).

Let's have a look at how that works:

```
@Testpublic void whenFilterEmployees_thenGetFilteredStream() {

   Integer[] empIds = { 1, 2, 3, 4 };


   List<Employee> employees = Stream.of(empIds)

     .map(employeeRepository::findById)

     .filter(e -> e != null)

     .filter(e -> e.getSalary() > 200000)

     .collect(Collectors.toList());


   assertEquals(Arrays.asList(arrayOfEmps[2]), employees);}
```

In the example above, we first filter out *null* references for invalid employee ids and then again apply a filter to only keep employees with salaries over a certain threshold.

findFirst

findFirst() returns an *Optional* for the first entry in the stream; the *Optional* can, of course, be empty:

```
@Testpublic void whenFindFirst_thenGetFirstEmployeeInStream() {

    Integer[] empIds = { 1, 2, 3, 4 };


    Employee employee = Stream.of(empIds)

      .map(employeeRepository::findById)

      .filter(e -> e != null)

      .filter(e -> e.getSalary() > 100000)

      .findFirst()

      .orElse(null);


    assertEquals(employee.getSalary(), new Double(200000));}
```

Here, the first employee with the salary greater than 100000 is returned. If no such employee exists, then *null* is returned.