Streams

Overview

In this lab you'll use Java 8 streams to perform operations on a collection of objects. This will also give you another opportunity to practise using lambda expressions, method references, and Java 8 standard functional interfaces.

Source folders

Student project:	`StudentStreams`
Solution project:	`SolutionStreams`

Roadmap

There are 8 exercises in this lab, of which the last two exercises are "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Familiarization
2. Using lambda expressions with stream operations
3. Mapping stream elements to a different type
4. Implementing a pipeline of operations
5. Filtering a stream by using a predicate
6. Accumulating and sorting elements
7. (If Time Permits) Performing matching and finding operations
8. (If Time Permits) Additional suggestions

Exercise 1: Familiarization

Open the *Student* project and take a look at `Employee.java`. Note the following points:

- Each employee has a unique ID, plus a name, office, and salary.
- We've provided a couple of helper methods to display the employee and to return a string representation of the employee.
- We've also implemented the `Comparable<Employee>` interface, which allows employees to be ordered by descending salary (i.e. higher-paid employee first).

Now take a look at `Main.java`. Note the following points:

- We create a sample collection of employees at class level, so we can use the collection in all the methods in the class.
- The class defines various methods to perform operations on the collection. We've implemented `displayEmployeeFullDetails()` already, to give you an idea what these methods will look like. In this method, we get a stream on the collection, and then call `forEach()` to apply an operation on each element. `forEach()` takes a parameter that implements `Consumer<T>`; to satisfy this requirement, we pass in a reference to the `Employee::display` instance method. The net effect is that all the employees will be displayed on the console.
- In `main()`, we invoke `displayEmployeeFullDetails()`.

Run `Main.java`, and verify all the employees are displayed on the console.

Exercise 2: Using lambda expressions with stream operations

In `Main.java`, implement `displayEmployeeNames()` to display the name of every employee on the console. Here are some hints:

- First, get a stream on the collection of employees.
- Call `forEach()` on the stream, to apply an operation on each employee. Use a lambda expression to specify the operation (the operation must get an employee's name and display it on the console).

In `main()`, add a call to your `displayEmployeeNames()` method. Then run the program and verify it displays all the employee names.

Exercise 3: Mapping stream elements to a different type

In `Main.java`, implement `displayWageBill()` to display the total of all the employees' salaries. Here are some hints:

- First, get a stream on the collection of employees.
- Call `mapToDouble()` on the stream, to map each `Employee` object to its salary (because it's only the employee's salary we're interested in). To achieve this effect, pass `Employee:getSalary` into the `mapToDouble()` function.
- Call `sum()` to sum all the values.

In `main()`, add a call to your `displayWageBill()` method. Then run the program and verify it displays the correct result (i.e. £2865000.00).

Exercise 4: Implementing a pipeline of operations

In `Main.java`, implement `displaySortedDistinctOffices()` to display a distinct list of all the offices for the employees, sorted alphabetically (i.e. *Berlin*, *Geneva*, *London*). Here are some hints:

- First, get a stream on the collection of employees.
- Call `map()` on the stream, to map each `Employee` object to just its office.
- Call `distinct()` on the stream, to eliminate duplicate values.
- Call `sorted()` on the stream, to sort values (the default order is ascending alphabetic).
- Finally, call `forEach()` to display all the values.

In `main()`, add a call to your `displaySortedDistinctOffices()` method. Then run the program and verify it displays all the distinct offices in ascending alphabetic order.

Exercise 5: Filtering a stream by using a predicate

In `Main.java`, take a look at `displayFilteredEmployees()`. The purpose of this method is to display all employees that satisfy a predicate. The method receives two parameters:

- A `String` message, describing the filter operation to be performed.
- A `Predicate<Employee>` object that specifies the test operation for the filter.

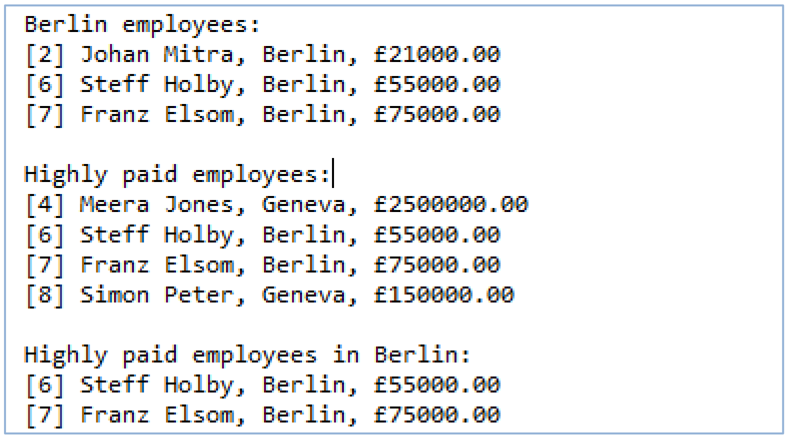
Implement `displayFilteredEmployees()` as follows:

- First, get a stream on the collection of employees.
- Call `filter()` on the stream, passing in the supplied predicate.
- Call `forEach()` to display the filtered employees.

In `main()`, add several separate calls to `displayFilteredEmployees()`. In each call, pass in a suitable predicate to achieve the following filtering:

- Berlin employees
- Highly paid employees (e.g. employees who earn more than 50000)
- Highly paid employees in Berlin

Then run the program and verify it displays the following results:



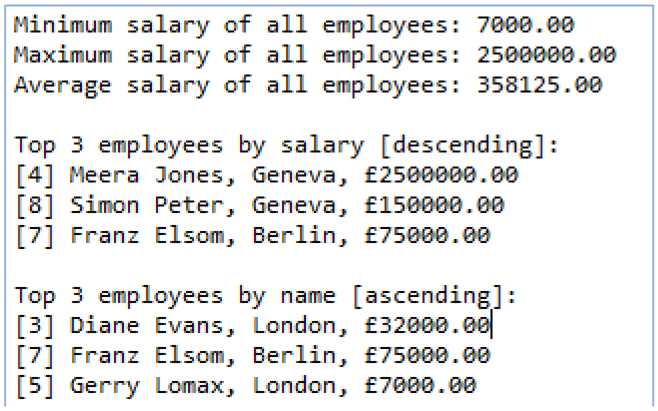
```
Berlin employees:
[2] Johan Mitra, Berlin, £21000.00
[6] Steff Holby, Berlin, £55000.00
[7] Franz Elsom, Berlin, £75000.00

Highly paid employees:
[4] Meera Jones, Geneva, £2500000.00
[6] Steff Holby, Berlin, £55000.00
[7] Franz Elsom, Berlin, £75000.00
[8] Simon Peter, Geneva, £150000.00

Highly paid employees in Berlin:
[6] Steff Holby, Berlin, £55000.00
[7] Franz Elsom, Berlin, £75000.00
```

Exercise 6: Accumulating and sorting elements

In `Main.java`, take a look at `displaySalaryStats()`. The purpose of this method is to display statistical information about employees in the collection. Implement the method so that it displays the following information:

- Minimum salary of all employees
- Maximum salary of all employees
- Average salary of all employees
- Top 3 employees by salary (in descending order, i.e. highest-paid first)
- Top 3 employees by name (in ascending alphabetic order)

Call this method from `main()` and then run the program. You should get the following results:



```
Minimum salary of all employees: 7000.00
Maximum salary of all employees: 2500000.00
Average salary of all employees: 358125.00

Top 3 employees by salary [descending]:
[4] Meera Jones, Geneva, £2500000.00
[8] Simon Peter, Geneva, £150000.00
[7] Franz Elsom, Berlin, £75000.00

Top 3 employees by name [ascending]:
[3] Diane Evans, London, £32000.00
[7] Franz Elsom, Berlin, £75000.00
[5] Gerry Lomax, London, £7000.00
```

Exercise 7 (If Time Permits): Performing matching and finding operations

In `Main.java`, take a look at `displaySalaryTests()`. The purpose of this method is to perform various matching and finding operations on employees in the collection. Implement the method so that it displays the following information:

- Do all employees earn at least the minimum wage (e.g. 7000)?
- Does any employee earn too much (e.g. 1000000)?
- Full details for the first employee in the specified city, or a suitable message if there is no employee in that city.

Call this method from `main()` and then run the program. Verify you get the correct results.

Exercise 8 (If Time Permits): Additional suggestions

Add some code to make use of the `reduce()` and `collect()` stream methods. You can find more information about these methods here:

- https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html