# Kotlin Arrays and Collections

## Overview

In this lab you'll use various features of the Kotlin collection API, focussing in particular on functional programming techniques.

## Roadmap

There are 6 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Define a data class and create a list of instances
2. Perform an operation on each item in a list
3. Map collection items to a different type
4. Implement a pipeline of operations
5. Filter a collection by using a predicate
6. (If Time Permits) Accumulate and sort elements

## Exercise 1: Define a data class and create a list of instances

Define an Employee data class containing the following info:

* The employee's ID (an integer)
* The employee's name (a string)
* The office where the employee works (a string)
* The employee's salary (a floating-point number)

The class should implement the Java Comparable<T> interface, so that you can compare Employee instances. The comparison should be based on salary, in ascending order.

Kotlin data classes automatically have a default toString method generated by the compiler. You can define your own custom toString method if you want to take control over formatting. Do that now, so that an employee might be displayed like this:

[1] Peter Smith, London, £25000.00

where [1] is the employee's ID in this example.

Write some client code to create a list of employees. Populate the list with about 8 sample employees with various salaries and offices. Some of the employees should work in the same offices, e.g. in the solution code all the employees work in London, Geneva, or Bergen.

## Exercise 2: Perform an operation on each item in a list

In this exercise you'll make use of the forEach method to perform an operation on each item in a collection. The forEach method is defined in the Iterable interface, which all Kotlin collection classes extend, so you can use forEach on any type of Kotlin collection.

So let's get started. Define a function named displayEmployeeFullDetails, which takes a list of employees as a parameter. Inside the function, use forEach to display each employee on the console. Call displayEmployeeFullDetails from your client code, then run your application and verify the full employee details are displayed correctly.

Now define another function named displayEmployeeNames along similar lines, except that it just displays the name of each employee. Call this function from your client code and verify it works as expected.

## Exercise 3: Map collection items to a different type

In this exercise you'll make use of the map method to transform collection items. The map method is defined as follows in the Iterable interface:

inline fun <T, R> Iterable<T>.map(

transform: (T) -> R

): List<R>

As you can see, map takes a *transform* function as a parameter. The function receives an item from the collection and returns a value of a (potentially) different type. After map has finished, it returns a list containing all the transformed values.

So define a function named displayWageBill as follows:

* The function should take a list of employees as a parameter.
* The function should use map to map each employee to its salary. What you'll get back is a List<Double>.
* Upon this List<Double>, call sum() to calculate the total wage bill for all the employees. The sum method is defined in Iterable – look it up in KotlinDocs to see the details.
* Display the total wage bill on the console.

Call displayWageBill from your client code and verify it works as expected.

## Exercise 4: Implement a pipeline of operations

In this exercise you'll chain together a pipeline of operations on a collection. Each operation returns a transformed collection, and acts as the target for the next operation in the pipeline.

Define a function named displaySortedDistinctOffices to display a distinct list of all the offices for the employees, sorted alphabetically (e.g. *Bergen*, *Geneva*, *London*). Here are some hints:

* The function should take a list of employees as a parameter.
* Call map on the collection, to map each Employee object to just its office. You'll end up with a collection of strings (i.e. offices).
* Call distinct to eliminate duplicate offices. Note that the distinct method is defined in the Iterable interface.
* Call sorted to sort the offices (the default order for strings is ascending alphabetic). Note that the sorted method is also defined in the Iterable interface.
* Finally, call forEach to display all the values.

Call displaySortedDistinctOffices from your client code and verify it displays all the distinct offices in ascending alphabetic order.

Now define a function named displayEmployeesSortedBySalary to display employees sorted by descending salary (i.e. higher earners first, lower earners last). To do this, call the standard sortByDescending method like this:

employees.sortedByDescending{it.salary}

Call displayEmployeesSortedBySalary from your client code and verify it displays employees in order of descending salary. For example, here's what it looks like in our solution code:

## 

## Exercise 5: Filter a collection by using a predicate

In this exercise you'll use the filter method to filter items in a collection. The filter method is defined as follows in the Iterable interface:

inline fun <T> Iterable<T>.filter(

predicate: (T) -> Boolean

): List<T>

As you can see, filter takes a *predicate* function as a parameter. The predicate function receives an item from the collection and returns true or false, indicating whether the item should be included (true) or excluded (false) from the result collection.

So, define a function named displayFilteredEmployees that takes 3 parameters:

* A string message, describing what kind of filtering is to be done (you'll pass in a suitable string message from the client code when you call this function).
* A list of employees.
* A predicate function, which specifies the test operation for the filter.

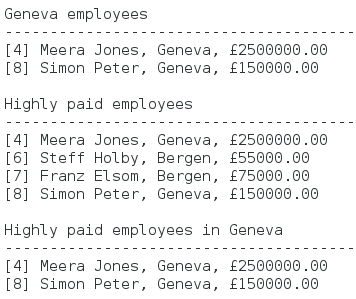
Implement displayFilteredEmployees as follows:

* Call filter on the collection, passing in the supplied predicate.
* Call forEach to display the filtered employees.

In the client code, calls displayFilteredEmployees several times. In each call, pass in a suitable predicate to achieve the following filtering (for example):

* Geneva employees
* Highly paid employees (e.g. employees who earn more than 50000)
* Highly paid employees in Geneva

Then run the application and verify it displays results such as the following:



## Exercise 6 (If Time Permits): Accumulate and sort elements

Define a function named displaySalaryStats. The purpose of this function is to display statistical information about employees in the collection. Implement the function so that it displays the following information:

* Minimum salary of all employees
* Maximum salary of all employees
* The top 5 employees by salary (in descending order, i.e. highest-paid first). You can use the take method to grab the first 5 items in the sorted collection.
* The top 5 employees by name (in ascending alphabetic order)

Call displaySalaryStats from your client code and then run the application. Verify you get the correct results.