

# Collections and Stream API

## \*Exercise 1

Imagine you are a computer scientist at a research and development center (R&D). This week, you are responsible to study the performance of some collection implementations in the JDK.

The collection classes are chosen for this test

include: `ArrayList`, `Vector`, `LinkedList`, `HashSet` and `PriorityQueue`.

You will develop a test program that measures performance for the following operations on the chosen collections:

Measure the time needed for adding 1 million, 2 million and 5 million elements. Using the method `add(Object)`.

Measure the time needed for adding one more element (the 1,000,001th element; 2,000,001th element; 5,000,001th element, respectively).

Measure the time needed to check the 500,005<sup>th</sup> element in a collection using the method `contains(Object)`.

Measure the time needed to remove the 800,008<sup>th</sup> element in a collection using the method `remove(Object)`.

Finally, this test program should generate the following reports:

### \* Performance Test Report #1: Adding elements

Collection	1 million	2 millions	5 millions
ArrayList	331 ms	2867 ms	13890 ms
Vector	551 ms	5230 ms	45210 ms
LinkedList	4900 ms	10100 ms	52001 ms
HashSet	5700 ms	12871 ms	67188 ms
PriorityQueue	6023 ms	33711 ms	88716 ms

### \* Performance Test Report #2: Adding one more element

Collection	1,00,001	2,000,001	5,000,001
ArrayList	331 ms	2867 ms	13890 ms
Vector	551 ms	5230 ms	45210 ms
LinkedList	4900 ms	10100 ms	52001 ms
HashSet	5700 ms	12871 ms	67188 ms
PriorityQueue	6023 ms	33711 ms	88716 ms

### \* Performance Test Report #3: Checking the 500,005th element and removing the 800,008<sup>th</sup> element

Collection	Check 500,005	Remove 8,000,008
ArrayList	331 ms	13890 ms
Vector	551 ms	45210 ms
LinkedList	4900 ms	52001 ms
HashSet	5700 ms	67188 ms
PriorityQueue	6023 ms	88716 ms

## \*Exercise 2

Imagine that you are an application developer for a renowned hospital in New York city. Your role is to develop a software application for managing customers queue at the hospital's reception room.

Upon startup, the application shows the following menu:

```
Welcome to the Hospital of New York!
```

```
New Customer
```

```
Next Customer
```

```
Print Waiting Line
```

```
Exit
```

```
Enter the option you would like to proceed (1-4): _
```

\* If the user chooses the option 1 (type '1' and then hit Enter in the command line interface), it asks for the name of the customer:

```
Enter customer's name: _
```

If the name already exists in the customer waiting queue, print the following message:

```
The customer is already in queue. Queue number is
```

If the name does not exist, add it to the queue and print the following message:

```
The customer has been queued. Queue number is
```

\* If the user chooses the option 2 (Next Customer), print the queue number of the next customer in the waiting line:

```
Next customer's number:
```

Then the application asks for the customer's name:

```
Enter customer's name: _
```

If the user enters the name corresponding to the queue number, the application removes the customer from the waiting line and print the following message:

```
The customer is being served.
```

If the name does not match the queue number, the application shows the following message:

```
Sorry, customer name does not match.
```

\* If the user chooses the option 3, the application prints all the customers are currently waiting in the following manner:

This list should be sorted by arrival time in ascending order, meaning that the first customer comes appears first in the list. Here's an example:

```
1      John Doe      19:21:00
2      Alice Smith 19:54:20
3      Carol Beck   20:01:15
4      David Hunt   20:16:28
```

If the queue is empty, print the following information:

```
There's no customers are currently waiting.
```

\* If the user chooses the option 4, the application prints the following message:

```
Thank you for coming to the Hospital of New York!
Good bye and See you gain.
```

Then the application terminates.

**NOTE:** When the application completes processing for an option, it shows the mane again to the user.

The `Customer` class is defined as the following:

```
public class Customer {
    protected String name;
    protected int queueNumber;
    protected java.util.Date arrivalTime;
}
```

### \*Exercise 3

We have the `Student` class as following:

```
import java.util.*;

public class Student {
    protected String name;
```

```

protected List< Subject > subjects = new ArrayList< >();

public Student() {
}

public Student(String name) {
    this.name = name;
}

public void addSubject(Subject subject) {
    this.subjects.add(subject);
}

public List< Subject > getSubjects() {
    return this.subjects;
}

public void setName(String name) {
    this.name = name;
}

public String getName() {
    return this.name;
}
}

```

**A student has a list of subjects. The `Subject` class is defined as following:**

```

public class Subject {
    protected String name;
    protected int mark;
    public Subject() {
    }
    public Subject(String name, int mark) {
        this.name = name;
        this.mark = mark;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}

```

```

    }

    public void setMark(int mark) {
        this.mark = mark;
    }

    public int getMark() {
        return this.mark;
    }
}

```

You are required to find at least 3 ways for printing average mark of each student like the following example:

```

Alex - 80.0
Beck - 84.6
Catherine - 77.1
David - 92.5

```

Here's my suggestion for the first 3 solution:

First solution: Use for loops without Stream API.

Second solution: Use Stream API by implementing the `getAverageMark()` method in the `Student` class.

Third solution: Use Stream API with just one statement.

## \*Exercise 4

- Evolve the library to able to compare two collections of any types that are sub types of `Item`.

- The super type `Item` is defined as following:

```

public interface Item {
    public int getId();
    public void setId(int id);
    public boolean isDifferent(Item copy);
}

```

- The method `isDifferent()` compares if two `Item` objects having different attributes or not (they have same id). This method will be used by the comparator class to identify a collection of updated items.

- Implement an abstract class called `AbstractItem` which is the base class for all objects which can be compared by the comparator:

```
public abstract class AbstractItem implements Item {  
    }  
}
```

- Hence the `Person` class should be re-defined like this:

```
public class Person extends AbstractItem {  
    }  
}
```

- The `PersonComparable` interface should be re-defined like this:

+ interface name: `ItemComparable`

+ methods: `getRemovedItems()`, `getNewItems()` and `getUpdatedItems()`

- Develop the `ItemComparator` class implements the `ItemComparable` interface so that it is able to compare two collections of any types which are sub types of the `Item`.

- Implement the `Book` class based on the following data:

```
public class Book extends AbstractItem {  
    String title;  
    float price;  
    int rating;  
}  
}
```

- Implement the `Car` class based on the following data:

```
public class Car extends AbstractItem {  
    String model;  
    String manufacturer;  
    int numberOfSeats;  
}  
}
```

- Write the `PersonComparatorTest` class that uses the `ItemComparator` to compare two collections of `Person` objects.

- Write the `BookComparatorTest` class that uses the `ItemComparator` to compare two collections of `Book` objects.

- Write the `CarComparatorTest` class that uses the `ItemComparator` to compare two collections of `Car` objects.