

May 16<sup>th</sup>, 2022

# Sheet 06 – Collections, Comparator, Comparable, Iterator

In this exercise sheet we will practice the usage collections, comparators, and iterators.

## Exercise 1 (Fibonacci Iterator)

[3 Points]

The goal of this exercise is to implement the Fibonacci numbers with iterators. Create the class `FibonacciIterator` that implements the `Iterator` interface. Your `FibonacciIterator` needs to provide a constructor where a long parameter `n` is passed as an argument, which determines the length of the fibonacci series your iterator has to provide (i.e. `n=10` means that your iterator delivers all fibonacci numbers from `fib(0)` to `fib(10)`).

### Hint





Using iterators is a requirement to successfully complete this exercise. Your implementation should be able to handle (theoretically) open ended number of fibonacci numbers within the limits of the JVM memory. You may find the `BigInteger` type useful.

Provide meaningful JUnit-Tests to show that your implementation works as expected. Make sure to cover all implemented methods within your tests.

### Submit



 `at/ac/uibk/pm/gXX/zidUsername/s06/e01/FibonacciIterator.java`

 `at/ac/uibk/pm/gXX/zidUsername/s06/e01/FibonacciIteratorTest.java`

## Exercise 2 (Election System)

[3 Points]

Implement a simple election system. The system should be built according to OOP principles based on the following requirements:

- An election system contains one or more voter regions and two or more political parties. All political parties compete in all voter regions.
- An electoral region contains a name and a maximum number of seats to be assigned.
- A political party has a name and total number of votes received from each voter region.

- An election system must have a `vote` method that distributes the number of votes to each party for each electoral region. This `vote`-method additionally requires a `VoteGenerator`-parameter, which implements a method to assign votes to each given party in each electoral region. Now implement a `RandomizedVoteGenerator`, which assigns randomly generated votes (range from 0 to 100000) to each given party in each electoral region.

Create an `ElectionApp` class with a main method. Test your implementation with minimum three different parties and three different regions. Call the `vote` method and print the resulting vote distribution.

Make sure that you choose the appropriate collection types from the Java Collections framework whenever it is necessary in the election system implementation.

#### Submit



```

at/ac/uibk/pm/gXX/zidUsername/s06/e02/ElectionSystem.java
at/ac/uibk/pm/gXX/zidUsername/s06/e02/ElectoralRegion.java
at/ac/uibk/pm/gXX/zidUsername/s06/e02/PoliticalParty.java
at/ac/uibk/pm/gXX/zidUsername/s06/e02/VoteGenerator.java
at/ac/uibk/pm/gXX/zidUsername/s06/e02/RandomizedVoteGenerator.java
...
```

### Exercise 3 (d'Hondt Method)

[4 Points]

In an election system, the votes are converted into the number of seats won by the political parties via various methods. The d'Hondt Method is a popular one. The method distributes the seats in an electoral region to the parties iteratively, based on the quotients of each party. Every party starts with the total number of votes received from an electoral region as their quotient. The party with the highest quotient wins one seat and its quotient is recalculated for the next iteration. At any given iteration, the quotient  $q_p$  of a seat winning party  $p$  is calculated based on the formula below:

$$q_p = \frac{V_p}{s_p + 1} \quad (1)$$

where  $V_p$  is the total number of votes a party received from that electoral region and  $s_p$  is the number of seats that political party has been assigned in that political region until that iteration. The iteration is done until all seats are distributed. See the example on the Wikipedia page for a better understanding how the process works<sup>1</sup>.

In this exercise, you are asked to implement the d'Hondt Method for the election system you developed in Exercise 2.

- Extend the `PoliticalParty` class with a property to represent the number of seats received in the parliament from each voter region.
- Implement the necessary interface in the `PoliticalParty` class and implement the `compareTo` method defined in that interface in order to enable sorting political parties alphabetically by name.
- Implement an `assignSeats` method on `ElectionSystem`. For each electoral region, the method should assign seats to the political parties by applying the d'Hondt Method explained above.

<sup>1</sup>[https://en.wikipedia.org/wiki/D%27Hondt\\_method](https://en.wikipedia.org/wiki/D%27Hondt_method)

- Implement two methods which return the political parties:
  - According to their names' alphabetical order (ascending order).
  - According to the number of seats they won (descending order).
- Implement a `FixedVoteGenerator` which assigns votes to each given party in each given region according to a fixed schema.
- Provide meaningful JUnit-Tests for the `assignSeats`-method and the methods which sort the political parties.




#### Hint



Your `FixedVoteGenerator` may be useful for your tests hence it allows you to determine the expected seats distribution.

#### Submit



 `at/ac/uibk/pm/gXX/zidUsername/s06/e03/FixedVoteGenerator.java`  
 `at/ac/uibk/pm/gXX/zidUsername/s06/e03/ElectionSystemTest.java`  
 `...`

**Important:** Submit your solution to OLAT and mark your solved exercises with the provided checkboxes. The deadline ends at 6:00 pm (18:00) on the day before the discussion.