Lab 9 - PARTI

3. In the lecture demo `lesson9.lecture.comparators2.EmployeeInfoBetter`, we showed how to use `Comparator.comparing` and `Comparator.thenComparing` to create better, more readable, and more functional-style `Comparator`s. In the demo code, however, there is branching logic that could be replaced by a cleaner design:

```
public void sort(List<Employee> emps, final SortMethod method) {
   if(method == SortMethod.BYNAME) {
      Collections.sort(emps,
          Comparator.comparing(byName)
                    .thenComparing(bySalary));
   } else {
      Collections.sort(emps,
          Comparator.comparing(bySalary)
                    .thenComparing(byName));
   }
}
```

Eliminate the branching logic by defining a `HashMap`, together with a `Pair` class, in a clever way. Start with the `EmployeeInfoBetter` and `Employee` classes from `lesson9.lecture.comparators2`, and then modify `EmployeeInfo` in a clever way that eliminates branching logic.

NOTE from Tina:
Try to do it by yourself, if you couldn't do this one, it's fine. I won't count this in the homework points. If you do it, there's no extra points for this one. So it's your choice.
For this homework, you need to create anothe class Pair which can make name, salary method to be two different pairs.
Then in the HashMap, you can store method with the pair, which means: if sorting method is SortMethod.BYNAME, then use name first, salary second Pair. If sorting method is SortMethod.BYSALARY, then use salary first, name second Pair.

The sort method will look like this:

```
@SuppressWarnings("unchecked")
public void sort(List<Employee> emps, final SortMethod method) {
    Collections.sort(emps,
            Comparator.comparing(sortDiscriminator.get(method).first)
                      .thenComparing(sortDiscriminator.get(method).second));

}
```

5. Create a method

```
Stream<String> streamSection(Stream<String> stream, int m, int n)
```

which extracts a substream from the input stream `stream` consisting of all elements from position `m` to position `n`, inclusive; you must use only `Stream` operations to do this. You can

assume `0 <= m <= n`. A Java class has been provided for you in the lab folder for this lesson; implement the method `streamSection` given in that class, and test using the `main` method provided.

NOTE from Tina:

Find correct method to do this from slides.