Objectives

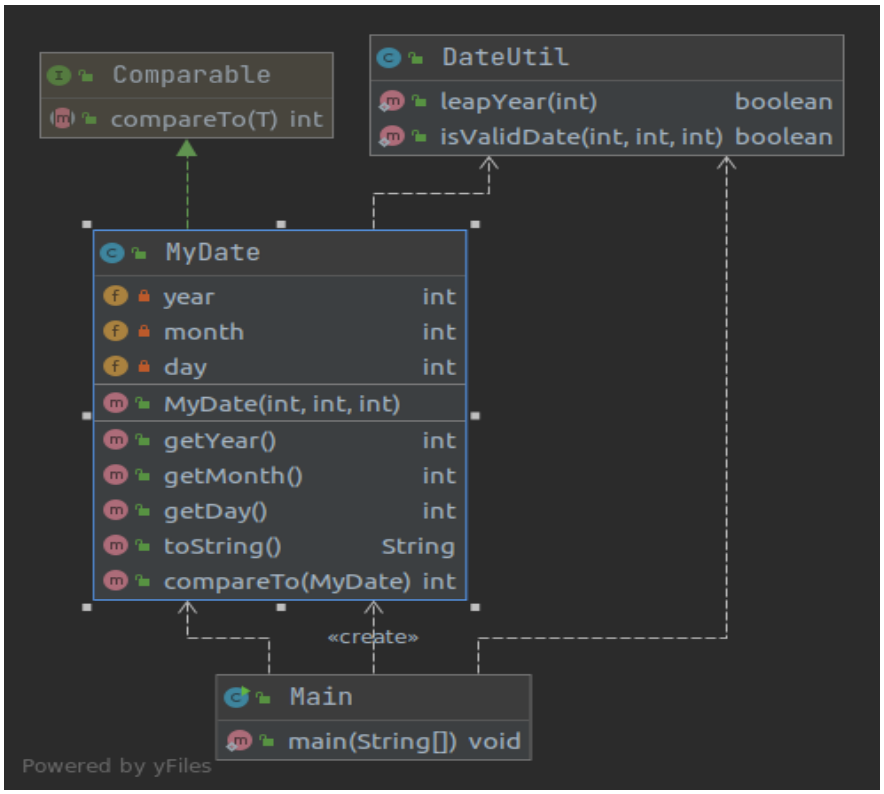
- Using interfaces
- Sorting:
  - Natural ordering: `Comparable<T>`
  - `Comparator<T>`

Create a Project/Module with 3 packages: `lab9_1, lab9_2, lab9_3`

Exercise 1.

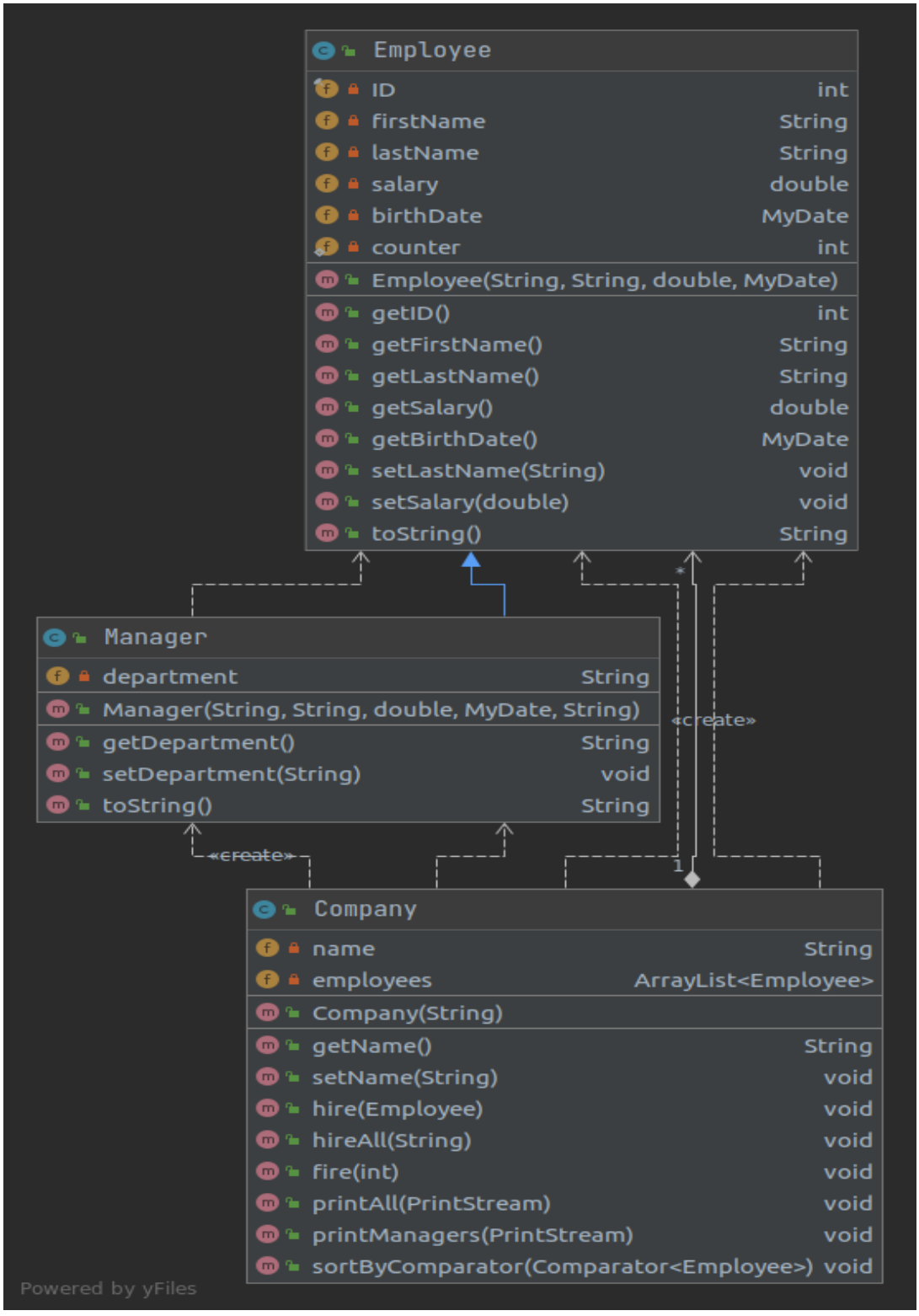
In this exercise you have to modify exercise 3 from Lab 2 (lab2_3 package).

- Create a copy of the `DateUtil` and the `MyDate` classes.
- Add a **natural ordering** to the `MyDate` class in order to be able to compare two dates (implement the Comparable<MyDate> interface!).
- Main class - main method:
  - Generate 10 valid dates from the current year and store them in an ArrayList
  - Print the ArrayList to the standard output
  - Sort the ArrayList
  - Print the ArrayList to the standard output



Exercise 2.

Implement the classes on the following class diagram:



Explanation:

- `hireAll (String csvFile)`: hires all the employees read from
- `sortByComparator( Comparator<Employee> comp)`: sorts the employees using the comp object
- `Input file example: employees.csv`

```
Incze, Zsolt-Tamas, 3100, 2000, 9, 25, WebDev
Kacso, Robert, 2900, 1998, 11, 15
Dumbravean-Katai, David, 3200, 2000, 8, 25
Balint, Zsolt, 1500, 1998, 2, 1
Fuzi, Zalan, 2100, 1999, 7, 9
Horvath, Janos, 2500, 1999, 12, 1
Bagoly, Norbert, 3000, 1999, 10, 1
Gabos, Alpar, 2900, 1997, 6, 1
Burszan, Hunor, 3500, 2000, 5, 3, MobileDev
```

- `sortByComparator` usage:

```
System.out.println("Alphabetically: ");
comp.sortByComparator(new Comparator<Employee>() {
   @Override
   public int compare(Employee e1, Employee e2) {
       ...
   }
});
comp.printAll(System.out);
```

- Sort the employees by the following criteria (one by one):
  - Alphabetically
  - Decreasing salary order
  - Managers followed by employees, both categories sorted alphabetically. Expected output for this criterion:

```
Manager{ID=9 ,firstName='Burszan', lastName='Hunor', ...}
Manager{ID=1 ,firstName='Incze', lastName='Zsolt-Tamas', ...}
Employee{ID=7, firstName='Bagoly', lastName='Norbert', ...}
Employee{ID=4, firstName='Balint', lastName='Zsolt', ...}
Employee{ID=3, firstName='Dumbravean-Katai', lastName='David', ...}
Employee{ID=5, firstName='Fuzi', lastName='Zalan', ...}
Employee{ID=8, firstName='Gabos', lastName='Alpar', ...}
```

```
Employee{ID=6, firstName='Horvath', lastName='Janos', ...}
Employee{ID=2, firstName='Kacso', lastName='Robert', ...}
```

Exercise 3.

Storage

A text file (`data.txt`) contains product data. Each line contains the following information: identifier (`int`), name (`String`), amount (`int`), price (`int`). Data in a given line is separated by whitespaces. Each identifier is unique.

Create a `Product` class which stores the data and has the following behaviour:

- constructor (4 parameters)
- getters (for each attribute)
- `increaseAmount(int newAmount)` method which increases the amount by newAmount
- `toString()`

Complete the class by adding a **natural ordering**, allowing ordering by `id` (increasing order).

Create a `Storage` class allowing the storage of multiple products. Besides the storage, the class has to allow quick updates of the stored products, therefore you should provide quick search based on product ID.
Add the following behaviour to the class:

- A `constructor` which has a filename as a parameter, and stores the products read from the file.
- An `update` method which has a filename as a parameter. The method reads the data from the file (`update.txt`) and updates all products read from the file. The structure of the update file is as follows: `id` and `newAmount` separated by whitespaces. This file also contains identifiers that are not in the storage. The method prints the number of products that were successfully updated.

Download the input files, run your program for each pair of input files and measure execution time! Complete the following table:

Products	Updates	Number of products updated	Execution time
data1000.txt	update1000.txt		
data1000.txt	update1000000.txt		
data1000000.txt	update1000.txt		
data1000000.txt	update1000000.txt		