PS Programming Methodology

University of Innsbruck - Department of Computer Science

Blaas A., Ernst M., Frankford E., Huaman Quispe E., Hupfauf B., Kaltenbrunner L., Moosleitner M., Priller S., Simsek U.

universität innsbruck
Department of Computer Science

May 23th, 2022

Sheet 07 – Generics

Exercise 1 (Generic inMemory database) [3 Points]

In this exercise you have to provide a rudimentary implementation of an inMemory database. `InMemory` means, that you do not have to worry about a persistence mechanism to store your database entries, but rather you can store them inside adequate `Java-Collections`. Your database needs to be generic, i.e. implemented once properly, it should be reusable for all kind of types (classes; denoted in the following as `entities`). There is only one restriction to such a type: It requires a `unique identifier` (primary key). This unique identifier is not restricted to a type, i.e. it can be a `Long`, `String` or any other type.

a) *2 Points* Now implement the following methods. Please note that the (generic) type of the mentioned `unique identifier` is denoted as `U` and the (also generic) type of the stored `entity class` is denoted as `T`:

- `T save(T entity)`: Saves an entity to the database and returns the saved object. If the unique identifier used for the entity is already contained within the database, an adequate exception is thrown.
- `T delete(T entity)`: Deletes an entity from the database and returns the deleted object. If the entity does not exists within the database, an adequate exception is thrown.
- `Optional<T> findOne(U id)`: Finds and returns an entity by its unique identifier (id) as `Optional`. Why is it better to use an Optional as return-value? What are Optionals in Java for?
- `List<T> findAll(Comparator<T> comparator)`: Returns all entities stored within the database. They need to be sorted according to the given `comparator`.

Submit

`at/ac/uibk/pm/gXX/zidUsername/s07/e01/InMemoryDatabase.java`

`...`

Hint

You may need additional classes to implement the exercise correctly. Please add them to the required-files-configuration-list on your own if you perform the automated submission check.

b) *1 Point* Now implement a set of `JUnit`-Tests to show that your implementation works as expected. Test each method at least once and consider also checking if Exceptions are thrown properly. In order to show that your generic implementation works as expected, test your database-implementation with the following entities:

- `User`: Has the properties `username` (String; unique identifier), `name` (String) and `surname` (String).
- `Item`: Has the properties `id` (Long; unique identifier), `name` (String) and `price` (double).

Submit

- `at/ac/uibk/pm/gXX/zidUsername/s07/e01/User.java`
- `at/ac/uibk/pm/gXX/zidUsername/s07/e01/Item.java`
- `at/ac/uibk/pm/gXX/zidUsername/s07/e01/UserDatabaseTest.java`
- `at/ac/uibk/pm/gXX/zidUsername/s07/e01/ItemDatabaseTest.java`

Exercise 2 (Generic tournament manager) [5 Points]

This exercises' goal is to implement a generic tournament-manager, where playing entities (players, teams; denoted as "players" for the sake of simplicity) compete against each other in a tournament. The entities which are participating initially are scheduled into different `groups`. Once the groups are scheduled, `encounters` between the entities within each group are generated (e.g. playerA competes against playerB in group 1). When the encounters are generated, the competition starts. For each encounter among all groups a `competition-result` is generated. The competition-result may looks like:

`playerA` vs. `playerB` `resultPlayerA` : `resultPlayerB`

Please note that the `result` should not be restricted to integer numbers, but rather be generic (e.g. it can be time like in a race, a string-based rating which shows that one player performed better than the other or a more complex one like for e.g. the hit/miss ratio of something). There is only one restriction: There must exist some kind of order between the result-type, i.e. it must be determinable which score is better in order to identify, which player won and which lost).
Now by beeing able to decide which player won/lost, it is required to assign points to a victory, draft and loss (e.g. 3 points for the winning player, 1 for both players at draft, 0 for the losing player). Once points are assigned, all results of all encounters within all groups can be processed in order to create a `ranking-list`. For each group such a ranking-list is created and based on that list, the two leading players are qualified to reach the final stage of the tournament.
When implementing the `TournamentManager`, consider the following requirements:

- Your implementation needs to be able to deal with multiple "competing entities" (e.g. teams, players, cars,...). Only restriction: They need to have a `name` property.
- You need to implement functionality to assign all your participating players to `groups`. The number of groups within the tournament needs to be configurable. The assignment-method gets a `GroupGenerator` as parameter, which is a component that takes care of creating the groups and assigning players to them. You also need to provide a `RandomGroupGenerator`, which randomly distributes the participating players among all teams.

Hint ⚠

Try to make the `GroupGenerator` exchangeable, i.e. assure that this component can be easyly exchanged (e.g. with a `GroupGenerator` which does not make a random distribution, but uses a predefined configuration).

- By knowing all players within the group, your implementation must provide functionality to generate all encounters between all players within the group. Every player must compete against evey other player at least once.
- Once you have generated all encounters, they need to be ”evaluated”: Evaluation means, that results are assigned to the two players participating at the encounter. This may look like:

  `playerA` vs. `playerB` 3:1

  An evaluation of an encounter is performed in two steps:

  1. A score is assigned to both players (e.g. playerA gets the score 3, playerB the score 1 as described in the example). Your evaluation method requires a `ResultGenerator` as parameter, which creates scores for both players of the encounter. You have to provide a `RandomResultGenerator` which randomly generates scores.

     Hint ⚠

     Again, try to make your `ResultGenerator` easyly exchangeable.

  2. Based on the assigned scores it is required to determine which player won, lost or if a draft happened. Please consider that the determination of the winning/losing player needs to be able to deal with generic result-types, i.e. not only with numbers.

     Hint ⚠

     Your result-determination method may need a `Comparator` as parameter in order to compare scores in a flexible way.

- After all of the encounters are evaluated, the players’ points (remember: 3 point for a winning player, 0 for a losing player, 1 for draw) are summarized and a final ranking-list is created for each group. Provide a method which creates a ranking-list out of an evaluated set of encounters.

Submit

- at/ac/uibk/pm/gXX/zidUsername/s07/e02/Player.java
- at/ac/uibk/pm/gXX/zidUsername/s07/e02/Group.java
- at/ac/uibk/pm/gXX/zidUsername/s07/e02/GroupGenerator.java
- at/ac/uibk/pm/gXX/zidUsername/s07/e02/RandomGroupGenerator.java
- at/ac/uibk/pm/gXX/zidUsername/s07/e02/ResultGenerator.java
- at/ac/uibk/pm/gXX/zidUsername/s07/e02/RandomResultRandomGenerator.java
- at/ac/uibk/pm/gXX/zidUsername/s07/e02/TournamentManager.java
- ...

Exercise 3 (Generic tournament simulator) [2 Points]

This exercise is based on the previous one and uses the implemented `TournamentManager` to simulate a tournament. Please reuse your `TournamentManager` and implement the following process:

1. Create a set of players.
2. Create groups and assign players to the groups randomly by using your `RandomGroupGenerator`.
3. For each Group:
   - Create all possible encounters.
   - Evaluate all encounters by using your `RandomResultGenerator`.
   - Create a ranking-list and determine the x leading players of each group (x depends on your group-size, so chose a senseful value).
4. Create a new group with the x leading players of each group (i.e. the `final group` which contains all x leading players of each group).
5. Create the encounters, evaluate them and create a final ranking-list.
6. Print the final ranking list. It needs to have at least the following information:
   - Rank
   - Name of the player
   - Number of encounters the player participated
   - Number of wins/draws/losses
   - Number of points

Submit

- at/ac/uibk/pm/gXX/zidUsername/s07/e03/TournamentSimulator.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.