Введение в программирование на Java
Домашнее задание 3

04 июня 2025

Описание задания

В рамках данного домашнего задания требуется реализовать работу небольшого языка запросов (а-ля SQL).

Важное замечание №1: проверка корректности реализации будет проводиться автоматически на наборе тестов; **любые** отклонения от формата ввода-вывода (даже лишние пробелы) считаются ошибкой.

Важное замечание №2: в рамках этого ДЗ вам дан шаблонный проект (находится в Smart LMS), части которого необходимо реализовать; **любые** отклонения от шаблона считаются ошибкой. Если не сказано явно, код менять запрещается (в коде шаблона указаны соответствующие комментарии). Если это не нарушает запретов, то добавлять новые классы, интерфейсы и методы разрешается (и рекомендуется).

База данных

В рамках ДЗ мы будем эмулировать базу данных (при помощи класса `Database`).

В реальных базах данных может быть множество таблиц, состоящих из различных колонок. В рамках этого ДЗ мы будем хранить только пользователей (класс `User`), каждый из которых будет иметь:

- Уникальный идентификатор (генерируется автоматически) — `id`.
- Имя (непустая строка) — `firstName`.
- Фамилия (непустая строка) — `lastName`.
- Родной город (непустая строка) — `city`.
- Возраст (неотрицательное число) — `age`.

Саму базу данных будем эмулировать обычным списком.

Таким образом, в шаблоне реализованы два класса, **которые запрещается изменять**:

- `User` с полями (и их getter-методами):
  - `id`
  - `firstName`
  - `lastName`
  - `city`
  - `age`
- `Database` с методами:
  - `List<User> getAll()` — неизменяемый список всех пользователей.
  - `int add(User user)` — добавление пользователя в БД; возвращает `id` нового пользователя.
  - `boolean remove(int id)` — удаление пользователя по `id`; возвращает `false`, если такого пользователя не существует, и `true` в случае успешного удаления.
  - `int size()` — количество пользователей в БД.
  - `void clear()` — очистка БД.

Общая архитектура приложения

Приложение должно парсить запросы из командной строки, обрабатывать их и выводить результат.

Код класса `Main` реализован за вас, его менять **запрещается**.

Класс `QueryParser` занимается парсингом команд (см. их список ниже). В рамках ДЗ требуется его реализовать. В классе разрешается менять всё, кроме:

- Самого определения класса (в т.ч. имя, модификаторы, родительские классы и интерфейсы).
- Сигнатуры, возвращаемого типа и модификаторов метода `parse`.

`QueryParser::parse` парсит команду из строки и возвращает результат в виде экземпляра класса `ParsingResult<Query>`.

Класс `ParsingResult` — результат парсинга, который может быть в одном из двух состояний:

- Успешный парсинг. Создаётся статическим методом `ParsingResult::of` и передачей объекта-запроса типа `Query`.
- Ошибка парсинга. Создаётся статическим методом `ParsingResult::error` и передачей сообщения об ошибке.

В класс `ParsingResult` разрешается добавлять новые вспомогательные методы (по аналогии с `map`), не нарушающие логику класса. Любые другие изменения в этом классе запрещены.

Интерфейс `Query` представляет собой запрос, который можно выполнить над базой данных (методом `execute`).

Интерфейс `QueryResult` представляет собой результат выполнения запроса. Интерфейс имеет метод `message`, который должен возвращать строку для печати в консоль.

Список поддерживаемых команд

В программе должны поддерживаться следующие команды:

- `SELECT` (с подкомандами `FILTER` и `ORDER`) — выборка данных.
- `INSERT` — вставка данных.
- `REMOVE` — удаление данных.
- `CLEAR` — очистка БД.

Все команды и подкоманды — регистронезависимые (т.е. "SELECT", "SeLeCt", "select" должны восприниматься как одно и то же). Ради простоты далее они указаны в верхнем регистре.

Во всех командах пробелы должны требоваться ровно в тех местах, где указаны в синтаксисе (причём могут быть в произвольном количестве). Пробелы в любых других местах не должны допускаться (внимательнее смотрите на `FILTER` и `ORDER`).

Команда выборки

Команда позволяет произвести выборку данных из БД.

Синтаксис:

```
SELECT (%ИМЯ_ПОЛЯ%, %ИМЯ_ПОЛЯ%, ..., %ИМЯ_ПОЛЯ%)
SELECT *
```

`%ИМЯ_ПОЛЯ%` соответствует имени поля из класса `User`, т.е. может принимать следующие значения:

- `id`
- `firstName`
- `lastName`
- `city`
- `age`

Например:

```
SELECT (firstName, lastName)
```

Выбранные данные должны выводиться через запятую+пробел, по одной строке на каждого пользователя. **Никаких других лишних символов быть не должно (в т.ч. пробелов и символов перевода строки)!**

Имена полей регистрозависимые (например, `FIRSTNAME` должен считаться некорректным именем).

В рамках одной команды допускается выбрать одно и то же поле несколько раз. Т.е. следующий запрос валиден:

```
SELECT (age, age, age, age, age, age, age, age, age, age)
```

Если же вместо имён полей использован символ `*`, то должны выбираться все поля класса `User` в порядке их объявления. Т.е. следующие запросы эквивалентны:

```
SELECT *
SELECT (id, firstName, lastName, city, age)
```

Пример работы (зелёное — пользовательский ввод, чёрное — вывод программы):

```
SELECT *
0, Ivan, Ivanov, Moscow, 18
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
SELECT (firstName)
Ivan
Petya
San
SELECT (age, firstName, age, id)
18, Ivan, 18, 0
30, Petya, 30, 1
30, San, 30, 2
```

В рамках команды выборки может быть указана команда фильтрации. Синтаксис:

```
SELECT (%ИМЯ_ПОЛЯ%, %ИМЯ_ПОЛЯ%, ..., %ИМЯ_ПОЛЯ%) FILTER(%ИМЯ_ПОЛЯ%, %ЗНАЧЕНИЕ%)
SELECT * FILTER(%ИМЯ_ПОЛЯ%, %ЗНАЧЕНИЕ%)
```

`FILTER` должен выбирать только тех пользователей, у которых значение поля равно указанному значению. Например, следующий запрос выбирает имена тех пользователей, которые живут в Москве:

```
SELECT (firstName) FILTER(city, Moscow)
```

`FILTER` может встречаться произвольное количество раз — в таком случае должны выбираться данные, удовлетворяющие одновременно всем фильтрам.

Пример работы (зелёное — пользовательский ввод, чёрное — вывод программы):

```
SELECT *
0, Ivan, Ivanov, Moscow, 18
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
SELECT * FILTER(age, 30)
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
SELECT (id) FILTER(age, 30) FILTER(firstName, Petya)
1
```

В рамках команды выборки может быть указана команда сортировки. Синтаксис:

```
SELECT (%ИМЯ_ПОЛЯ%, %ИМЯ_ПОЛЯ%, ..., %ИМЯ_ПОЛЯ%) ORDER(%ИМЯ_ПОЛЯ%, %ASC|DESC%)
SELECT * ORDER(%ИМЯ_ПОЛЯ%, %ASC|DESC%)
```

`ORDER` упорядочивает по возрастанию (`ASC`) или убыванию (`DESC`) указанного поля (для строк применяется лексикографический порядок). В случае, если значения поля, по которому идёт упорядочивание, имеют дубликаты, то эти дубликаты дополнительно упорядочиваются по возрастанию поля `id`.

Если `ORDER` отсутствует, то упорядочивание производится по возрастанию поля `id`.

`ORDER` встречается не более одного раза.

`FILTER` и `ORDER` могут встречаться в любом порядке. Например:

```
SELECT * FILTER(city, Moscow) ORDER(age, ASC) FILTER(lastName, Ivanov)
```

Пример работы (зелёное — пользовательский ввод, чёрное — вывод программы):

```
SELECT *
0, Ivan, Ivanov, Moscow, 18
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
SELECT * ORDER(age, DESC)
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
0, Ivan, Ivanov, Moscow, 18
SELECT * ORDER(city, ASC)
0, Ivan, Ivanov, Moscow, 18
2, San, Sanych, Samara, 30
1, Petya, Petrov, St.Petersburg, 30
SELECT (id) ORDER(city, ASC) FILTER(age, 30)
2
1
```

Классы:

- `SelectQuery` — исполняемый запрос на выборку (реализует интерфейс `Query`).
  - Класс **запрещается изменять**, кроме содержимого метода `execute` — его нужно реализовать.
  - Поле `getters` — список функций, которые переводят пользователя в строковое значение поля.
    - Функции представлены функциональным интерфейсом `FieldGetter`.
    - При парсинге создаётся экземпляр этого интерфейса для каждого из указанных полей. Т.е. при запросе `SELECT (firstName)` должен быть создан один экземпляр `FieldGetter`, возвращающий поле `firstName` для передаваемых ему пользователей.
    - Интерфейс **запрещается изменять**, но вам потребуется создать его реализации при помощи лямбд/анонимных классов/ссылок на методы.
  - Поле `predicate` — предикат, по которому фильтруются пользователи (результат обработки `FILTER`).
  - Поле `comparator` — компаратор для сортировки пользователей (результат обработки `ORDER`).
- `SelectQueryResult` — результат выборки (реализует интерфейс `QueryResult`).
  - Класс **запрещается изменять**, кроме содержимого метода `message` — его нужно реализовать.
  - Поле `selectedValues` — список строковых значений выбранных полей для каждого пользователя (т.е. результат выборки, фильтрации и упорядочивания).
  - Метод `message` должен вернуть одну единственную строку, которая будет напечатана в консоль (см. описание и скриншоты с форматом выше).

Команда вставки

Команда позволяет добавить новую запись о пользователе в БД.

Синтаксис:

```
INSERT (%ИМЯ%, %ФАМИЛИЯ%, %ГОРОД%, %ВОЗРАСТ%)
```

Например:

```
INSERT (Ivan, Ivanov, Ivanovo, 34)
```

В качестве строковых значений должны допускаться любые символы, кроме символа запятой. Пустые строки и строки, состоящие только из пробельных символов не должны допускаться. Пробельные символы в начале и в конце строк должны удаляться — записи в БД попадают без этих пробелов.

В качестве возраста должны допускаться только неотрицательные целые числа.

Пример работы (зелёное — пользовательский ввод, чёрное — вывод программы):

```
SELECT *
0, Ivan, Ivanov, Moscow, 18
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
INSERT (Ivan, Ivanov, Ivanovo, 34)
User with id 3 was added successfully
SELECT *
0, Ivan, Ivanov, Moscow, 18
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
3, Ivan, Ivanov, Ivanovo, 34
```

Классы:

- `InsertQuery` — исполняемый запрос на вставку (реализует интерфейс `Query`).
  - Класс **запрещается изменять**, кроме содержимого метода `execute` — его нужно реализовать.
  - Поле `user` — добавляемый пользователь. Идентификатор данного пользователя может быть произвольным числом — он будет автоматически установлен классом `Database`.
- `InsertQueryResult` — результат вставки (реализует интерфейс `QueryResult`).
  - Класс **запрещается изменять**, кроме содержимого метода `message` — его нужно реализовать.
  - Поле `id` — идентификатор добавленного пользователя.

Команда удаления

Команда позволяет удалить запись о пользователе по идентификатору из БД.

Синтаксис:

```
REMOVE %ИДЕНТИФИКАТОР%
```

Например:

```
REMOVE 42
```

Если пользователя с указанным идентификатором не существует, то команда должна корректно парситься, но её выполнение должно возвращать сообщение об ошибочном удалении.

Пример работы (зелёное — пользовательский ввод, чёрное — вывод программы):

```
SELECT *
0, Ivan, Ivanov, Moscow, 18
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
REMOVE 1
User with id 1 was removed successfully
SELECT *
0, Ivan, Ivanov, Moscow, 18
2, San, Sanych, Samara, 30
```

Классы:

- `RemoveQuery` — исполняемый запрос на удаление (реализует интерфейс `Query`).
  - Класс **запрещается изменять**, кроме содержимого метода `execute` — его нужно реализовать.
  - Поле `id` — идентификатор удаляемого пользователя.
- `RemoveQueryResult` — результат удаления (реализует интерфейс `QueryResult`).
  - Класс **запрещается изменять**, кроме содержимого метода `message` — его нужно реализовать.
  - Поле `id` — идентификатор удаляемого пользователя.
  - Поле `success` — существовал ли пользователь перед удалением.

Команда очистки

Примечание: эта команда уже реализована в шаблоне и может использоваться как пример.

Команда удаляет все данные из БД.

Синтаксис:

```
CLEAR
```

Пример работы (зелёное — пользовательский ввод, чёрное — вывод программы):

```
SELECT *
0, Ivan, Ivanov, Moscow, 18
1, Petya, Petrov, St.Petersburg, 30
2, San, Sanych, Samara, 30
CLEAR
3 users were removed successfully
SELECT *
```

Классы (оба запрещено изменять):

- `ClearQuery` — исполняемый запрос на очистку (реализует интерфейс `Query`).
- `ClearQueryResult` — результат очистки (реализует интерфейс `QueryResult`).

Обработка некорректных входных данных

Все некорректные входные данные (например, выборка несуществующего поля, отсутствующий пробел) должны быть корректно обработаны. При вводе некорректных данных программа должна сообщать об ошибке, но продолжать работать.

Для сообщения об ошибке должны использоваться следующие способы:

- Возврат `ParsingResult::error` из `QueryParser::parse`.
- Возврат строки с сообщением из метода `message` класса, реализующего интерфейс `QueryResult`.

Таким образом, самостоятельная **печать сообщения об ошибке в консоль не допускается** (это делается автоматически в классе `Main`).

Все возможные ситуации ошибочных данных заведомо не приводятся. Найти и правильно обработать такие входные данные: одна из задач данного домашнего задания.

Тестирование

Для программы должны быть реализованы модульные тесты с использованием **JUnit5**.

Инструкцию по настройке JUnit можно найти в репозитории курса. Неправильно настроенная библиотека ведёт к снижению оценки.

К оформлению модульных тестов применяются те же правила кодирования, что и ко всему остальному коду.

Помимо этого:

- Все модульные тесты должны что-то проверять (т.е. иметь вызов `assert...` или подобного метода для выполняемых действий). Не считается тестами код, который ничего не проверяет, а лишь делает видимость таковой проверки с целью увеличения покрытия.
- Все модульные тесты должны быть небольшими и тестировать одну конкретную вещь.
- Все модульные тесты должны быть независимыми — результат прохождения тестов не должен зависеть от порядка их запуска.

Правила кодирования

Программа должна быть реализована в объектно-ориентированном стиле с использованием функциональных интерфейсов.

Исходный код программы должен быть совместим с Java 21 (при разработке рекомендуется использовать OpenJDK 21).

Код **не должен** быть написан в одном единственном классе и/или методе. Программа должна быть декомпозирована на небольшие классы, интерфейсы и методы, каждый из которых решает свою задачу.

Классы должны инкапсулировать свои данные. Не должно быть возможности привести объект в некорректное состояние при помощи каких-то из его методов.

Каждый класс, интерфейс и прочие определения типов должны находиться в отдельных файлах, каждый в своём (исключением являются только статические и нестатические вложенные классы/интерфейсы/т.п., если они обоснованны).

Модификаторы доступа должны выбираться осознанно. Отсутствие модификатора доступа (т.е. `package-private` доступ) должно быть обоснованно. Не допускается делать все поля и методы публичными.

Все файлы с исходным кодом должны быть сгруппированы по пакетам. Файлы, не принадлежащие никакому пакету, не допускаются.

Прочие правила кодирования, в соответствии с которыми должна быть оформлена программа, можно найти в соответствующем документе.

Критерии оценивания

Оценка состоит из трёх компонент: реализация, тестирование и оформление.

- **Реализация (4 балла)**
  - 4 балла — описанный функционал полностью реализован, программа корректно обрабатывает все ошибочные ситуации.
  - Оценка понижается за частично реализованный функционал, неправильно обработанные ошибки или прочие отклонения от указанных требований.
  - Для каждого из варианта команд оценивается только его **полная** реализация. Т.е. если реализованы все классы-наследники `Query` и `QueryResult`, но не реализован парсинг команд — команды считаются нереализованными.
    - Рекомендация: реализуйте команды поэтапно, начиная с самых простых по синтаксису.
  - **Любые** отклонения от формата ввода-вывода (даже лишние пробелы) или от реализуемого шаблона считаются ошибкой.
- **Тестирование (3 балла)**
  - 3 балла — покрытие строк кода составляет 75% и более. Все тесты успешно проходят. Все тесты соответствуют требованиям к модульным тестам.
  - 2 балла — покрытие строк кода составляет 50-74%, либо при небольших отклонениях от требований к тестам (в т.ч. при неуспешных тестах).
  - 1 балл — покрытие строк кода составляет 25-49%, либо при сильных отклонениях от требований к тестам (в т.ч. при неуспешных тестах).
  - 0 баллов — покрытие строк кода составляет менее 25% или тесты вовсе отсутствуют.
- **Оформление (3 балла)**
  - 3 балла — код следует всем указанным правилам кодирования.
  - Оценка понижается за нарушение правил кодирования (в зависимости от регулярности и серьёзности).

Отдельные случаи:

- Если код программы не компилируется, выставляется оценка 0 (допускаются исключения на усмотрение проверяющего, если ошибка легко исправляется, а программа в целом корректно реализована).
- При обнаружении несамостоятельного выполнения задания все участники получают оценку 0 (в том числе и настоящий автор программы).

Формат сдачи задания

Задание должно быть загружено в **Smart LMS**.

Дедлайн: 17 июня 2025, 23:59.

В качестве решения должен быть приложен zip-архив, содержащий проект в IntelliJ IDEA с решением, в том числе:

- Директорию .idea и файл *.iml, находящиеся в корне проекта.
- Директорию src с исходными файлами *.java (**в т.ч. с файлами самого шаблона**).
- Директорию test с исходными файлами *.java, содержащими модульные тесты на основе JUnit.

При наличии деталей реализации, о которых хотелось бы уведомить проверяющего, рекомендуется создать текстовый файл README с необходимым описанием. Файл README прикладывается в архив.

Прочих "мусорных" файлов в архиве быть **не должно**.

Архив должен иметь имя `HW3_<ГРУППА>_<ФИО>.zip`, например `HW3_ББИ2401_ИвановИванИванович.zip`.

Нарушение правил именования архива, наличие в нём лишних файлов или отсутствие необходимых влечёт за собой **снижение оценки**.

Пример корректного архива:

```
— HW3_ББИ2401_ИвановИванИванович.zip
    — .idea
        — workspace.xml
        — misc.xml
        — modules.xml
        — ...
    — QueryLanguage.iml
    — README.txt
    — src
        — querylang
            — Main.java
            — ...
    — test
        — querylang
            — ...
```