4th homework assignment; OPRPP1

Napravite prazan Maven projekt: u Eclipsovom workspace direktoriju napravite direktorij `hw04-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.oprpp1.jmbag0000000000:hw04-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema `junit5`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka.

Problem 1.

Na Ferku je dostupan JAR imena `lsystems.jar`. Skinite ga, te ga instalirajte u Vaš lokalni mavenov repozitorij pod koordinatama hr.fer.oprpp1.fractals:lsystems:1.0. Potom u `pom.xml` Vaše zadaće dodajte ovisnost prema toj biblioteci). Ova biblioteka sadrži jezgru za vizualizaciju jedne vrste fraktala: *Lindermayerovih sustava*. S ovom vrstom fraktala možete se upoznati googleanjem, ili pak proučite u knjizi: http://java.zemris.fer.hr/nastava/irg/knjiga-0.1.2016-03-02.pdf
kratak opis u poglavlju 13.6.

U paketu `hr.fer.zemris.lsystems` već su dostupna sljedeća sučelja:

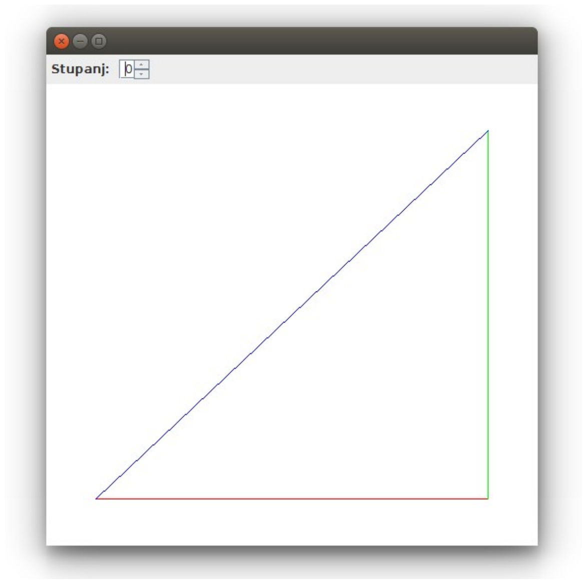
```
public interface LSystem {
	String generate(int level);
	void draw(int level, Painter painter);
}

public interface LSystemBuilder {
	LSystemBuilder setUnitLength(double unitLength);
	LSystemBuilder setOrigin(double x, double y);
	LSystemBuilder setAngle(double angle);
	LSystemBuilder setAxiom(String axiom);
	LSystemBuilder setUnitLengthDegreeScaler(double unitLengthDegreeScaler);
	LSystemBuilder registerProduction(char symbol, String production);
	LSystemBuilder registerCommand(char symbol, String action);

	LSystemBuilder configureFromText(String[] lines);

	LSystem build();
}

public interface LSystemBuilderProvider {
	LSystemBuilder createLSystemBuilder();
}

public interface Painter {
	void drawLine(
	  double x0, double y0, double x1, double y1, Color color, float lineWidth
	);
}
```

Sučelje `LSystem` modelira jedan Lindermayerov sustav. Metoda `generate` prima razinu i vraća string koji odgovara generiranom nizu nakon zadanog broja razina primjena produkcija. Ako je razina 0, vraća se aksiom, za razinu 1 vraća se niz dobiven primjenom produkcija na aksiom, odnosno općenito, ako je razina *k*, vraća se niz dobiven primjenom produkcija na niz dobiven za razinu *k*-1. Primijetite: ova metoda ništa ne crta: prima broj (razinu) i vraća jedan string koji predstavlja niz simbola koje sustav generira na toj razini.

Metoda `draw` uporabom primljenog objekta za crtanje linija koji je tipa `Painter` crta rezultantni fraktal. Konceptualno, može pitati metodu `generate` da izgenerira niz simbola za traženu razinu, i potom uoprabom objekta `Painter` taj niz simbola može, simbol po simbol, nacrtati.

Sučelje `Painter` modelira objekte kojima je moguće crtati linije po površini prozora. Ovi objekti nad prozorom razapinju koordinatni sustav čije je ishodište donji lijevi ugao, *x*-os ide udesno i kraj prozora ima *x*-koordinatu 1 (dakle, čitava os je na intervalu [0,1]), a *y*-os ide prema gore i vrh prozora ima *y*-koordinatu 1 (pa je i ova čitava os na intervalu [0,1]). Sljedeći jednostavan programčić "zlorabit" će sučelje `LSystem` kako bi na površini prozora nacrtao trokut (donji lijevi ugao je na 10% širine i visine prozora, donji desni na 90% širine i 10% visine, gornji desni ugao je na 90% širine i visine prozora) – stavite ga u neki demonstracijski program i stvarno pokrenite, da vidite što se događa. Objekt `Painter` koji u metodi `draw` dobiva `LSystem` automatski preračunava stvarne "integer" koordinate. Mijenjajte za probu mišem dimenzije prozora i uočite kako navedeni postotci uvijek ostaju očuvani.

```java
LSystemViewer.showLSystem(new LSystem() {
        @Override
        public String generate(int level) {
                return ""; // totalno ignoriramo u ovom primjeru...
        }
        @Override
        public void draw(int level, Painter painter) {
                painter.drawLine(0.1, 0.1, 0.9, 0.1, Color.RED, 1f);
                painter.drawLine(0.9, 0.1, 0.9, 0.9, Color.GREEN, 1f);
                painter.drawLine(0.9, 0.9, 0.1, 0.1, Color.BLUE, 1f);
        }
});
```


Stupanj: 0

Sučelje `LSystemBuilder` modelira objekte koje je moguće konfigurirati i potom pozvati metodu `build()` koja vraća jedan konkretan Lindermayerov sustav prema zadanoj konfiguraciji. Konfiguracija se može napraviti na dva načina: pozivanjem odgovarajućih metoda ili pak metode `configureFromText`.

Sučelje `LSystemBuilderProvider` modelira objekte koji znaju stvarati objekte za konfiguriranje Lindermayerovih sustava.

Evo dva primjera koji pojašnjavaju kako se ovo sve zajedno može koristiti. Oba pretpostavljaju da kao ulaz dobivaju objekt preko kojega mogu stvarati objekte za konfiguriranje Lindermayerovih sustava.

```java
private static LSystem createKochCurve(LSystemBuilderProvider provider) {
	return provider.createLSystemBuilder()
			.registerCommand('F', "draw 1")
			.registerCommand('+', "rotate 60")
			.registerCommand('-', "rotate -60")
			.setOrigin(0.05, 0.4)
			.setAngle(0)
			.setUnitLength(0.9)
			.setUnitLengthDegreeScaler(1.0/3.0)
			.registerProduction('F', "F+F--F+F")
			.setAxiom("F")
			.build();
}

private static LSystem createKochCurve2(LSystemBuilderProvider provider) {
	String[] data = new String[] {
			"origin                 0.05 0.4",
			"angle                  0",
			"unitLength             0.9",
			"unitLengthDegreeScaler 1.0 / 3.0",
			"",
			"command F draw 1",
			"command + rotate 60",
			"command - rotate -60",
			"",
			"axiom F",
			"",
			"production F F+F--F+F"
	};
	return provider.createLSystemBuilder().configureFromText(data).build();
}
```

Konfiguracija koja se temelji na nizu linija (posljednji primjer) podrazumijeva da svaka linija sadrži po jednu direktivu (ili je prazna). Direktive se sljedeće:

- `origin`: zadaje točku iz koje kornjača kreće; nad prozorom u kojem će biti prikazana slika bit će rastegnut virtualni koordinatni sustav čije je ishodište dolje lijevo, plus x-os ide prema desno i desni rub prozora ima x-koordinatu 1, plus y-os ide prema gore i vrh prozora ima y-koordinatu 1; centar prozora je tada (0.5, 0.5)
- `angle`: kut prema pozitivnoj osi-x smjera u kojem kornjača gleda; kut od 0 stupnjeva znači da kornjača gleda u desno; kut od 90° znači da kornjača gleda prema vrhu ekrana, itd.
- `unitLength`: koliko je dugačak jedinični pomak kornjače; primjerice, vrijednost 0.5 ako kornjača gleda u desno bi odgovaralo polovici širine prozora
- `unitLengthDegreeScaler`: kako bi se dimenzije prikazanog fraktala zadržale manje-više konstantnima, ako se generira niz za dubinu `d`, `unitLength` je potrebno na odgovarajući način skalirati; stoga je početnu efektivnu duljinu koraka za kornjaču potrebno postaviti na
  `unitLength * (unitLengthDegreeScaler^d)`
  vrijednost za ovaj broj može biti jedan decimalni broj, ili pak decimalni broj / decimalni broj (uz proizvoljan broj razmaka između svega što uključuje i nula razmaka)
- `command`: za simbol koji slijedi definira akciju koju kornjača mora napraviti; moguće akcije popisane su u nastavku; konfiguacija može sadržavati više ovih direktiva ali moraju biti za različite simbole
- `axiom`: početni niz iz kojeg kreće razvoj sustava; može biti samo jedan simbol ali može biti niz
- `production`: za simbol definira niz kojim se isti zamjenjuje; konfiguracija može sadržavati više ovih direktiva, ali moraju biti za različite simbole

Akcije koje treba podržati su sljedeće:

- "`draw` *s*": pomiče kornjaču u smjeru u kojem je okrenuta za s efektivne duljine pomaka i pri tome ostavlja trag trenutnom bojom; ažurira njezin položaj; primjer: "`draw 0.5`"
- "`skip` *s*": kao draw samo ne ostavlja trag (efekt je kao da je kornjača preskočila taj dio)
- "`scale` *s*": efektivnu duljinu pomaka ažurira tako da je pomnoži danim faktorom; npr. "`scale 0.75`"
- "`rotate` *a*": rotira smjer u kojem gleda kornjača za dani kut u stupnjevima (rotacija je u matematički pozitivnom smjeru); npr. "`rotate -90`"
- "`push`": trenutno stanje kornjače pohranjuje na stog i kopiju stavlja na vrh
- "`pop`": sa stoga vraća prethodno pohranjeno stanje kornjače
- "`color` *rrggbb*": ažurira boju kojom treba crtati na zadanu; npr. "`color ff0000`" aktivira crvenu boju

Prilikom obrade tekstovnih zapisa sve višestruke praznine i tabove treba zanemariti.

Vaš je zadatak sljedeći. Prilikom modeliranja opisanog u nastavku razmišljajte doslovno o kornjači koja se nalazi negdje na ekranu te se može kretati po njemu. Za kornjaču bismo, primjerice, željeli znati gdje se točno nalazi te u kojem je smjeru okrenuta – tako da, kada joj kažemo da se pomakne "naprijed" znamo odrediti što to točno znači (pomak u lijevo, desno, prema gore, dolje, ukoso, ...?). Stoga u paketu `hr.fer.zemris.lsystems.impl` definirajte razred `TurtleState` koji pamti trenutnu poziciju na kojoj se kornjača nalazi (`Vector2D`, ovdje vektor tretiramo kao radij-vektor koji čuva poziciju kornjače; razred iskopirajte iz prethodne zadaće), smjer u kojem kornjača gleda (`Vector2D`, jedinične duljine, ovo tretiramo kao klasični "vektor" koji čuva smjer), boju kojom kornjača crta (`Color`) te trenutnu efektivnu duljinu pomaka (`double`). Trenutna efektivna duljina pomaka odgovara duljini za koju će se kornjača pomaknuti ako joj se naredi jedinični pomak (primjerice, naredba `draw` ima argument koji govori koliko se u odnosu na jedinični pomak želimo pomaknuti; `draw 0.5` bi tražilo pomak duljine pola trenutne efektivne duljine). Definirajte javnu metodu `copy()` koja vraća novi objekt s kopijom trenutnog stanja. Ako se kopija na bilo koji način mijenja, to ne smije mijenjati ništa što se čuva u originalnom stanju pa pazite na to.

U istom paketu definirajte razred `Context`; primjerci ovog razreda omogućavaju izvođenje postupka prikazivanja fraktala te trebaju ponuditi stog na koji je moguće stavljati i dohvaćati stanja kornjače; za stog koristite Vaš razred `ObjectStack` (iskopirajte ga iz prethodne zadaće). Trenutno odnosno aktivno stanje kornjače je ono koje se nalazi na vrhu stoga. Mnoge naredbe izravno modificiraju to stanje (ne miču ga sa stoga niti ga dupliciraju; primjerice, naredba `color` će u trenutnom stanju promijeniti zapisanu boju).
U razredu `Context` napravite javne metode:

```
TurtleState getCurrentState(); // vraća stanje s vrha stoga bez uklanjanja
void pushState(TurtleState state); // na vrh gura predano stanje
void popState(); // briše jedno stanje s vrha
```

U istom paketu definirajte sučelje `Command`; sučelje treba definirati jednu metodu:

```
void execute(Context ctx, Painter painter);
```

Napravite novi paket `hr.fer.zemris.lsystems.impl.commands`. Svaku od naredbi koje kornjača mora biti u stanju raditi modelirajte jednim razredom smještenim u ovaj paket; razred mora implementirati sučelje `Command`. Napravite tako:

- `PopCommand`: briše jedno stanje s vrha stoga
- `PushCommand`: stanje s vrha stoga kopira i kopiju stavlja na stog
- `RotateCommand(angle)`: kroz konstruktor prima `angle`; u trenutnom stanju modificira vektor smjera gledanja kornjače tako što ga rotira za zadani kut
- `DrawCommand(step)`: kroz konstruktor prima `step`; računa gdje kornjača mora otići; povlači liniju

zadanom bojom od trenutne pozicije kornjače do izračunate i pamti u trenutnom stanju novu poziciju kornjače
- `SkipCommand(step)`: kao `DrawCommand` samo što ne povlači liniju
- `ScaleCommand(factor)`: kroz konstruktor prima `factor`; u trenutnom stanju ažurira efektivnu duljinu pomaka skaliranjem s danim faktorom
- `ColorCommand(Color)`: kroz konstruktor prima boju (kao primjerak razreda `java.awt.Color`); u trenutno stanje kornjače zapisuje predanu boju

U prethodnom tekstu pojam "u trenutno stanje" podrazumijeva stanje koje se iz konteksta dobije kao zapisano na vrhu – dakle metodom `Context#getCurrentState()`.

Konačno, u paket `hr.fer.zemris.lsystems.impl` smjestite `LSystemBuilderImpl` koji implementira `LSystemBuilder`. Razred `LSystemBuilderImpl` treba koristiti dva primjerka razreda `Dictionary` (iskopirajte iz prethodne zadaće): preko jednog treba pamtiti registrirane produkcije; preko drugoga registrirane akcije (tj. naredbe). *Napomena*: akcije se u Vašem primjerku razreda `Dictionary` koji ih pamti ne moraju pamtiti kao stringovi koje neprestano "parsirate" kad god Vam treba pojedina akcija (dapače, nemojte to raditi). Ovaj razred koristi i sljedeće pretpostavljene vrijednosti (uz prazne rječnike):

```
unitLength = 0.1;
unitLengthDegreeScaler = 1;
origin = new Vector2D(0, 0);
angle = 0;
axiom = "";
```

U metodi `build()` stvara se jedan konkretan `LSystem`; stoga je implementirajte tako da vrati primjerak Vašeg novog razreda (nekog ugniježđenog) koji implementira sučelje `LSystem`. Taj razred u metodi `draw()` stvara novi kontekst, stvara novo stanje kornjače (crna boja je default), gura ga na stog konteksta (čime to postaje trenutno stanje), potom za zadanu dubinu koju metoda `draw` prima kao argument poziva generiranje konačnog znakovnog niza za tu dubinu (metodu `generate`) i na kraju, za svaki znak tako dobivenog generiranog niza, iz riječnika dohvaća naredbu; ako naredbe nema, prelazi se odmah na sljedeći znak; ako naredba postoji, izvršava se (čime se ili modificira trenutno stanje, ili se nešto crta na zaslonu, ili se radi kopija trenutnog stanja i gura na stog, ili se sa stoga skida jedno stanje čime prethodno sa stoga postaje aktualno trenutno stanje) i zatim prelazi na sljedeći znak.

Dajmo primjer za jedan L-sustav čiji je Aksiom "`F`", te pravilo "`F -> F+F--F+F`". Za takav bi sustav vrijedilo sljedeće:

`generate(0)` vraća `F`
`generate(1)` vraća `F+F--F+F`
`generate(2)` vraća `F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F`

Napišite junit-testove baš za ovaj primjer i provjerite da vaša implementacija metode `generate` radi korektno.

Napravite program `Glavni1` smješten u paket `demo`. U metodu `main` stavite samo naredbu:

```
LSystemViewer.showLSystem(createKochCurve(LSystemBuilderImpl::new));
```

i dodajte metodu koju pozivamo, a dana je u ovoj uputi. Ako ste sve dobro napravili, po pokretanju ćete dobiti prozor gdje ćete moći mijenjati stupanj i vidjeti generiranu krivulju.

Napravite program `Glavni2` smješten u paket `demo`. U metodu `main` stavite samo naredbu:

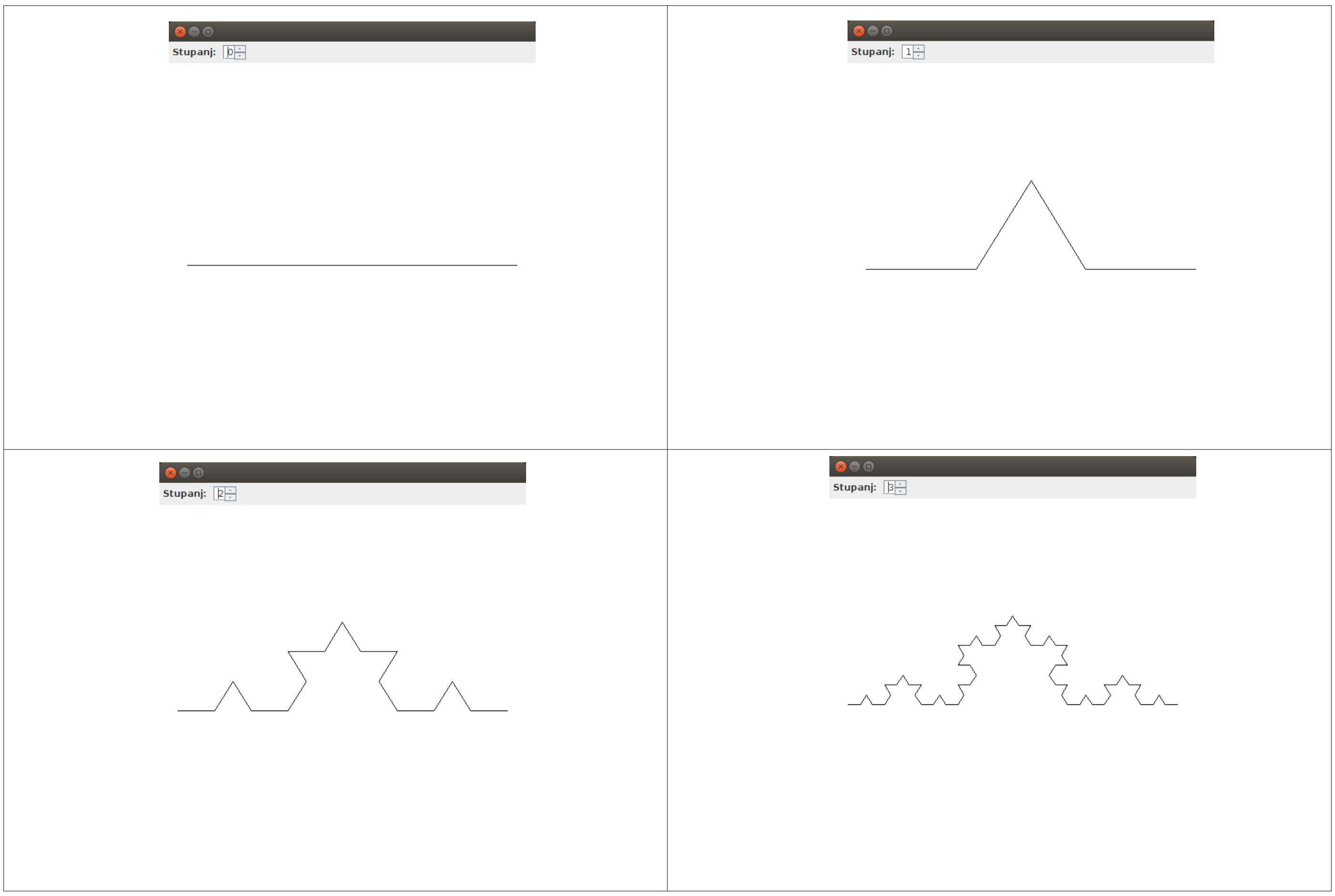
```
LSystemViewer.showLSystem(createKochCurve2(LSystemBuilderImpl::new));
```

i dodajte metodu koju pozivamo, a dana je u ovoj uputi. Ako ste sve dobro napravili, po pokretanju ćete dobiti prozor gdje ćete moći mijenjati stupanj i vidjeti generiranu krivulju. Primijetite da ovdje GUI koji sam pripremio interagira s Vašom implementacijom L-sustava. Pri otvaranju prozora, te svaki puta kada na vrhu prozora promijenite trenutni stupanj, GUI će se pobrinuti da obriše trenutnu sliku, pripremi novi `Painter` te nad Vašom implementacijom `LSystem` pozove metodu `draw` i preda joj podešeni stupanj te stvoreni `Painter`. Zadaća Vaše metode `draw` je pozivima metoda primljenog `Painter`-a nacrtati sliku fraktala koja odgovara odabranom stupnju.

Napravite program `Glavni3` smješten u paket `demo`. U metodu `main` stavite samo naredbu:

```
LSystemViewer.showLSystem(LSystemBuilderImpl::new);
```

Ako ste sve dobro napravili, po pokretanju ćete dobiti prozor gdje ćete moći iz tekstovne datoteke učitati definiciju fraktala te zatim mijenjati stupanj i vidjeti generirani fraktal.



Ako Vam se širina krivulje bude mijenjala, niste dobro implementirali opisan način izračuna efektivne duljine koraka.

Kao dodatna ZIP arhiva na Ferka je uploadan i niz tekstovnih datoteka s primjerima drugih fraktala koje bi Vaša implementacija sada morala biti u stanju prikazivati: isprobajte ih.

Problem 2.

Please read the whole problem description before you start coding – there are implementation details later in the text. In this problem you can use Java Collection Framework and appropriate collection implementations.

Write a simple database emulator. Put the implementation classes in package `hr.fer.oprpp1.hw04.db.` In repository on Ferko you will find a file named `database.txt`. It is a simple textual form in which each row contains the data for single student. Attributes are: *jmbag*, *lastName*, *firstName*, *finalGrade*. Name your program `StudentDB`. When started, program reads the data from current directory from file `database.txt`. If in provided file there are duplicate jmbags, or if finalGrade is not a number between 1 and 5, program should terminate with appropriate message to user.

Write a class `StudentRecord`; instances of this class will represent records for each student. Assume that there can not exist multiple records for the same student. Implement `equals` and `hashCode` methods so that the two students are treated as equal if jmbags are equal: use IDE support to automatically generate these two methods.

Write the class `StudentDatabase`: its constructor must get a list of `String` objects (the content of `database.txt`, each string represents one row of the database file). It must create an internal list of student records. Additionally, it must create *an index* for fast retrieval of student records when jmbag is known (use map for this). This constructor must check that previously mentioned conditions are satisfied (no duplicate jmbags, valid grades). Add the following two public methods to this class as well:

```
public StudentRecord forJMBAG(String jmbag);
public List<StudentRecord> filter(IFilter filter);
```

The first method uses index to obtain requested record in O(1); if record does not exists, the method returns `null`. In order to implement this, use `Map` from Java Collection Framework. Interfaces `List` and `Map` here are those from *Java Collection Framework*. In this problem, you are free to use collection implementations from `java.util` package.

The second method accepts a reference to an object which is an instance of functional interface `IFilter`:

```
public interface IFilter {
   public boolean accepts(StudentRecord record);
}
```

The method `filter` in `StudentDatabase` loops through all student records in its internal list; it calls `accepts` method on given filter-object with current record; each record for which `accepts` returns **`true`** is added to temporary list and this list is then returned by the `filter` method.

The system reads user input from console. You must support a single command: **query**. Here are several legal examples of the this command.

```
query jmbag="0000000003"
query   lastName  =    "Blažić"
query firstName>"A" and lastName LIKE "B*ć"
query firstName>"A" and firstName<"C" and lastName LIKE "B*ć" and jmbag>"0000000002"
```

Please observe that command, attribute name, operator, string literal and logical operator AND can be separated by more than one tabs or spaces. However, space is not needed between atribute and operator, and between operator and string literal. Logical operator AND can be written with any casing: AND, and, AnD etc is OK. Command names, attribute names and literals are case sensitive.

`query` command performs search in two different ways.

1. If `query` is given only a single attribute (which must be `jmbag`) and a comparison operator is `=`, the command obtains the requested student using the indexing facility of your database implementation in O(1) complexity.
2. For any other query (a single `jmbag` but operator is not `=`, or any query having more than one attribute), the command performs sequential record filtering using the given expressions. Filtering expressions are built using only `jmbag`, `lastName` and `firstName` attributes. No other attributes are allowed in query. Filtering expression consists from multiple comparison expressions. If more than one expression is given, all of them must be composed by logical `AND` operator. To make this homework solvable in reasonable time, no other operators are allowed, no grouping by parentheses is supported and allowed attributes are only those whose value is string. This should considerably simplify the solution. This command should treat jmbag attribute the same way it treats other attributes: the expression `jmbag="..."` should perform that comparison and nothing more (one could argue that such query could be executed in O(1) if we use index); do not do it here – it will complicate thing significantly.

String literals must be written in quotes, and quote can not be written in string (so no escapeing is needed; another simplification). You must support following seven comparison operators: `>`, `<`, `>=`, `<=`, `=`, `!=`, `LIKE`. On the left side of a comparison operator a field name is required and on the left side string literal. This is OK: `firstName="Ante"` but following examples are invalid: `firstName=lastName`, `"Ante"=firstName`.

When `LIKE` operator is used, string literal can contain a wildcard `*` (other comparisons don't support this and treat `*` as regular character). This character, if present, can occur at most once, but it can be at the beginning, at the end or somewhere in the middle). If user enters more wildcard characters, throw an exception (and catch it where appropriate and write error message to user; don't terminate the program).

Please observe that query is composed from one or more conditional expressions. Each conditional expression has field name, operator symbol, string literal. Since only operator `and` is allowed for expression combining, you can store the whole query in an array (or list) of simple conditional expressions.

Define a strategy[1] named `IComparisonOperator` with one method:

```
public boolean satisfied(String value1, String value2);
```

Implement concrete strategies for each comparison operator you are required to support (see `ComparisonOperators` below for details). Arguments of previous method are two string literals (not field names).

Create a class `ComparisonOperators` which offers following public static final variables (i.e. constants) of type `IComparisonOperator`:

- `LESS`
- `LESS_OR_EQUALS`
- `GREATER`
- `GREATER_OR_EQUALS`
- `EQUALS`
- `NOT_EQUALS`
- `LIKE`

You can initialize them directly or in static initializer block, to instances of some private static classes which implement `IComparisonOperator` or even with lambda expressions. This will allow you to write a code

1 See: http://en.wikipedia.org/wiki/Strategy_pattern as well as text in book (there is glossary; look up Strategy desing pattern)

like this:

```
IComparisonOperator oper = ComparisonOperators.LESS;
System.out.println(oper.satisfied("Ana", "Jasna"));  // true, since Ana < Jasna
```

In *like* operator, first argument will be string to be checked, and the second argument pattern to be checked.

```
IComparisonOperator oper = ComparisonOperators.LIKE;
System.out.println(oper.satisfied("Zagreb", "Aba*"));  // false
System.out.println(oper.satisfied("AAA", "AA*AA"));    // false
System.out.println(oper.satisfied("AAAA", "AA*AA"));   // true
```

Then define another strategy: `IFieldValueGetter` which is responsible for obtaining a requested field value from given `StudentRecord`. This interface must define the following method:

```
public String get(StudentRecord record);
```

Write three concrete strategies: one for each `String` field of `StudentRecord` class (i.e. one that returns student's first name, one that returns student's last name, and one that returns student's jmbag) – see `FieldValueGetters` class below.

Create a class `FieldValueGetters` which offers following public static final variables of type `IFieldValueGetter`:

- `FIRST_NAME`
- `LAST_NAME`
- `JMBAG`

You can initialize them directly or in static initializer block, to instances of some private static classes which implement `IFieldValueGetter` or even with lambda expressions. This will allow you to write a code like this:

```
StudentRecord record = getSomehowOneRecord();
System.out.println("First name: " + FieldValueGetters.FIRST_NAME.get(record));
System.out.println("Last name: " + FieldValueGetters.LAST_NAME.get(record));
System.out.println("JMBAG: " + FieldValueGetters.JMBAG.get(record));
```

Finally, model the complete conditional expression with the class `ConditionalExpression` which gets through constructor three arguments: a reference to `IFieldValueGetter` strategy, a reference to string literal and a reference to `IComparisonOperator` strategy. Add getters for these properties (and everything else you deem appropriate). If done correctly, you will be able to use code snippet such as the one given below:

```
ConditionalExpression expr = new ConditionalExpression(
  FieldValueGetters.LAST_NAME,
  "Bos*",
  ComparisonOperators.LIKE
);

StudentRecord record = getSomehowOneRecord();

boolean recordSatisfies = expr.getComparisonOperator().satisfied(
  expr.getFieldGetter().get(record),  // returns lastName from given record
  expr.getStringLiteral()             // returns "Bos*"
);
```

Write a class `QueryParser` which represents a parser of query statement and it gets query string through constructor (actually, it must get everything user entered **after** `query` keyword; you must skip this keyword). I would highly recommend writing a simple lexer and parser: now you now how to do it.

`QueryParser` must provide the following three methods.

`boolean isDirectQuery();`
  => Method must return `true` if query was of of the form `jmbag="xxx"` (i.e. it must have only one comparison, on attribute `jmbag`, and operator must be equals). We will call queries of this form *direct queries*.

`String getQueriedJMBAG();`
  => Method must return the string (`"xxx"` in previous example) which was given in equality comparison in direct query. If the query was not a direct one, method must throw `IllegalStateException`.

`List<ConditionalExpression> getQuery();`
  => For <u>all queries</u>, this method must return a list of conditional expressions from query; please observe that for direct queries this list will have only one element.

When faced with queries which are not direct, please *do not waste time on optimization*. For example, query such `jmbag="1" and jmbag="2"` is obviously always false, but we do not care. Query such `jmbag="1" and lastName>"A"` could be treated as “pseudo-direct” and checked in O(1) – but we do not care. Designing optimizer for queries is problem for itself and is not the topic of this homework.

Here is a simple usage example for `QueryParser`.

```
QueryParser qp1 = new QueryParser(" jmbag       =\"0123456789\"    ");
System.out.println("isDirectQuery(): " + qp1.isDirectQuery()); // true
System.out.println("jmbag was: " + qp1.getQueriedJMBAG()); // 0123456789
System.out.println("size: " + qp1.getQuery().size()); // 1

QueryParser qp2 = new QueryParser("jmbag=\"0123456789\" and lastName>\"J\"");
System.out.println("isDirectQuery(): " + qp2.isDirectQuery()); // false
// System.out.println(qp2.getQueriedJMBAG()); // would throw!
System.out.println("size: " + qp2.getQuery().size()); // 2
```

When implementing operators >, >=, <, <= use method `String#compareTo`. Also be aware that the comparison order of letters is determined by current locale setting from operating system (so if you do not have croatian locale as active one, do not be surprised if “Č” ends up somewhere after “Z”) - this is OK.

Create a class `QueryFilter` which implements `IFilter`. It has a single public constructor which receives one argument: a list of `ConditionalExpression` objects.

Now follows an example of interaction among previously defined classes and interfaces.

```
StudentDatabase db = …
QueryParser parser = new QueryParser(… some query …);
if(parser.isDirectQuery()) {
  StudentRecord r = db.forJMBAG(parser.getQueriedJMBAG());
  … do something with r …
} else {
  for(StudentRecord r : db.filter(new QueryFilter(parser.getQuery()))) {
    … do something with each r …
  }
}
```

Implementation details

Here follows package placement and names for some of the previously defined interfaces and classes.

```
hr.fer.oprpp1.hw04.db.StudentRecord
hr.fer.oprpp1.hw04.db.IComparisonOperator
hr.fer.oprpp1.hw04.db.ComparisonOperators
hr.fer.oprpp1.hw04.db.IFieldValueGetter
hr.fer.oprpp1.hw04.db.FieldValueGetters
hr.fer.oprpp1.hw04.db.ConditionalExpression
hr.fer.oprpp1.hw04.db.QueryParser
hr.fer.oprpp1.hw04.db.StudentDatabase
hr.fer.oprpp1.hw04.db.IFilter
hr.fer.oprpp1.hw04.db.QueryFilter
```

Recommended order of writing:

1. Write `StudentRecord`.
2. Write `IFilter`.
3. Write `StudentDatabase`. Test that `forJMBAG(…)` works. In test class, write two private classes implementing `IFilter`: one which always returns `true` and one which always return `false`. Test that database method `filter(…)` returns all records when given an instance of first class and no records when given an instance of second class. Instead of classes, you can use lambda expressions.
4. Write interface `IComparisonOperator` and class `ComparisonOperators` and test your operator implementations. Be carefull with *like* operator! (e.g. check is `"AAA" LIKE "AA*AA"`)
5. Write and test `IfieldValueGetter/FieldValueGetters`.
6. Write and test `ConditionalExpression`.
7. Write and test `QueryParser`.
8. Write and test `QueryFilter`.

When implementing *query* command please observe that this command does two different things: it determines which records should be written based on filtering condition, and then produces correct formatting. Implement it that way (as two separate methods). This way, if you wish, you can separately test the record retrieval, and separately test the output formatting (which could be implemented as a method which receives a list of records, and produces a list of strings – each string one row; the final list can then be sent to `System.out`); something like (in pseudocode):

```
List<StudentRecord> records = select_from_database_based_on_filtering_condition(…)
List<String> output = RecordFormatter.format(records);
output.forEach(System.out::println);
```

Here is an example of interaction with program, as well as expected output and formatting. Symbol for prompt which program writes out is “> ”.

```
> query jmbag = "0000000003"
Using index for record retrieval.
+============+=======+========+===+
| 0000000003 | Bosnić | Andrea | 4 |
+============+=======+========+===+
Records selected: 1

> query jmbag = "0000000003" AND lastName LIKE "B*"
+============+=======+========+===+
| 0000000003 | Bosnić | Andrea | 4 |
+============+=======+========+===+
Records selected: 1

> query jmbag = "0000000003" AND lastName LIKE "L*"
Records selected: 0

> query lastName LIKE "B*"
+============+===========+===========+===+
| 0000000002 | Bakamović | Petra     | 3 |
| 0000000003 | Bosnić    | Andrea    | 4 |
| 0000000004 | Božić     | Marin     | 5 |
| 0000000005 | Brezović  | Jusufadis | 2 |
+============+===========+===========+===+
Records selected: 4

> query lastName LIKE "Be*"
Records selected: 0

> exit
Goodbye!
```

Please observe that the table is automatically resized and that the columns must be aligned using spaces. The order in which the records are written is the same as the order in which they are given in database file.

If users input is invalid, write an appropriate message. Allow user to write spaces/tabs: the following is also OK:

```
query     lastName="Bosnić"
query lastName    ="Bosnić"
query lastName=   "Bosnić"
query lastName  =    "Bosnić"
```

Note: for reading from the file please use the following code snippet. In the example, we are reading the content of `prva.txt` located in current directory:

```
List<String> lines = Files.readAllLines(
    Paths.get("./prva.txt"),
    StandardCharsets.UTF_8
);
```

The program should terminate when user enters the `exit` command.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else, unless otherwise stated in specific problem in this homework). Additionally, for this homework you can not use any of Java Collection Framework classes which represent collections or its derivatives for problem 1 but can for problem 2. For problem 1 you can use your own collection implementations! Document your code!

All source files must be written using UTF-8 encoding. All classes, methods and fields (public, private or otherwise) must have appropriate javadoc.

You are expected to write junit tests defined explicitly in this document. You are advised to use junit tests for other classes and methods. Please see "*Recommended order of writing*" in problem 2.

When your homework is done, pack it in zip archive with name `hw04-0000000000.zip` (replace zeros with your JMBAG). Upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted.