4. homework assignment; JAVA, Academic year 2016/2017; FER

Napravite prazan Maven projekt, kao u 1. zadaći: u Eclipsovom workspace direktoriju napravite direktorij `hw04-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.zemris.java.jmbag0000000000:hw04-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema `junit:junit:4.12`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka.

Problem 1.

Napišite implementaciju razreda `SimpleHashtable<K,V>` parametriziranog parametrima `K` i `V`. Razred predstavlja tablicu raspršenog adresiranja koja omogućava pohranu uređenih parova (ključ, vrijednost). Parametar `K` je tip ključa, parametar `V` je tip vrijednosti. Postoje dva javna konstruktora: defaultni koji stvara tablicu veličine 16 slotova, te konstruktor koji prima jedan argument: broj koji predstavlja željeni početni kapacitet tablice i koji stvara tablicu veličine koja je potencija broja 2 koja je prva veća ili jednaka predanom broju (npr. ako se zada 30, bira se 32); ako je ovaj broj manji od 1, potrebno je baciti `IllegalArgumentException`. Ova implementacija koristit će kao preljevnu politiku pohranu u ulančanu listu. Stoga će broj uređenih parova (ključ, vrijednost) koji su pohranjeni u ovoj kolekciji moći biti veći od broja slotova tablice (pojašnjeno u nastavku). Implementacija ovog razreda ne dozvoljava da ključevi budu `null` reference; vrijednosti, s druge strane, mogu biti `null` reference.

Jedan slot tablice modelirajte javnim ugniježđenim statičkim razredom `TableEntry<K,V>` (razmislite zašto želimo da je razred statički i što to znači). Primjerci ovog razreda imaju člansku varijablu `key` u kojoj pamte predani ključ, člansku varijablu `value` u kojoj pamte pridruženu vrijednost te člansku varijablu `next` koja pokazuje na sljedeći primjerak razreda `TableEntry<K,V>` koji se nalazi u *istom slotu* tablice (izgradnjom ovakve liste rješavat ćete problem preljeva – situacije kada u isti slot treba upisati više uređenih parova). Ove sve članske varijable trebaju biti privatne. Za ključ mora postojati isključivo javni getter a za vrijednost i javni getter i javni setter. Ostali getteri i setteri ne smiju postojati u ovom razredu. Implementacijski, svaki primjerak razreda `SimpleHashtable<K,V>` interno održava polje referenci na glave ulančanih lista; čvor ulančane liste modelira je razredom `TableEntry<K,V>`. Obratite pažnju da nećete moći izravno alocirati polje parametriziranih referenci – spomenuli smo to na predavanju i rekli kako postupiti.

Ideju uporabe ovakve kolekcije ilustrira sljedeći kod.

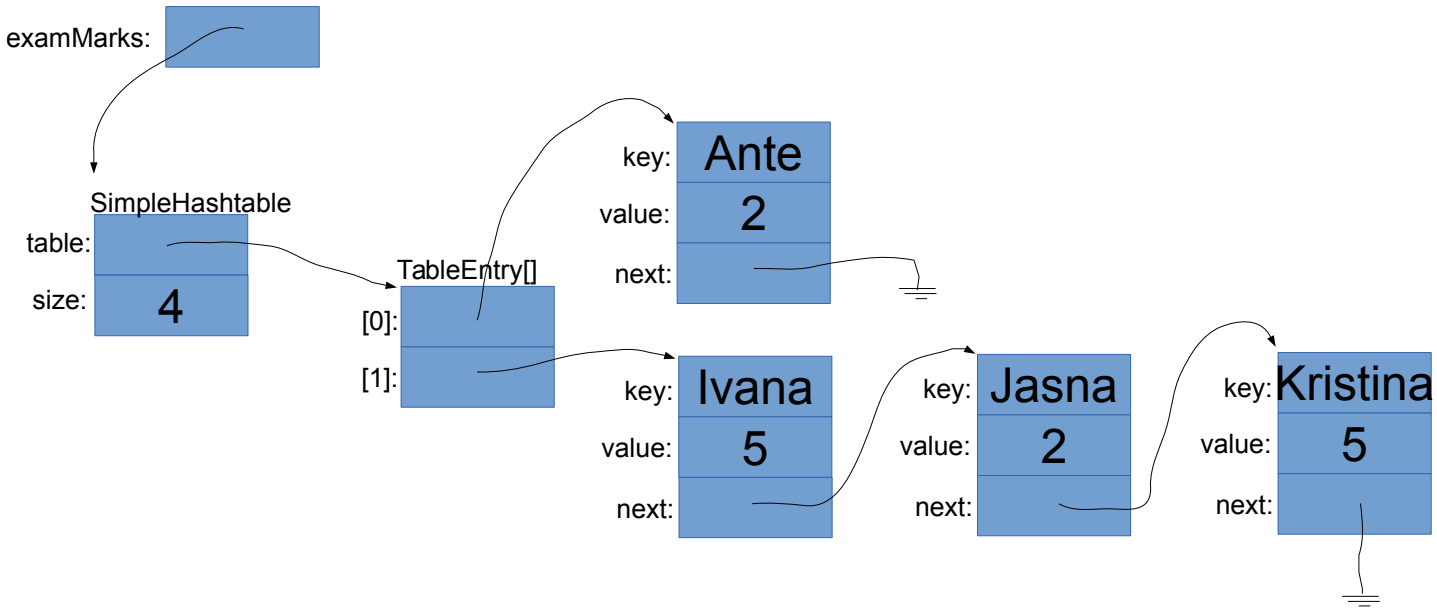
```
// create collection:
SimpleHashtable<String, Integer> examMarks = new SimpleHashtable<>(2);

// fill data:
examMarks.put("Ivana", 2);
examMarks.put("Ante", 2);
examMarks.put("Jasna", 2);
examMarks.put("Kristina", 5);
examMarks.put("Ivana", 5); // overwrites old grade for Ivana

// query collection:
Integer kristinaGrade = examMarks.get("Kristina");
System.out.println("Kristina's exam grade is: " + kristinaGrade); // writes: 5

// What is collection's size? Must be four!
System.out.println("Number of stored pairs: " + examMarks.size()); // writes: 4
```

Za potrebe izračuna slota u koji treba ubaciti uređeni par koristite metodu `hashCode()` ključa, pa modulo veličina tablice. Ključ uređenog para ne smije biti `null` dok vrijednost može biti `null`.

Razred `SimpleHashtable<K,V>` treba imati sljedeće članske varijable:

- `TableEntry<K,V>[] table`: polje slotova tablice,
- `int size`: broj parova koji su pohranjeni u tablici.

Pojednostavljeni prikaz stanja u memoriji nakon izvođenja koda iz prethodnog primjera ilustriran je na sljedećoj slici.



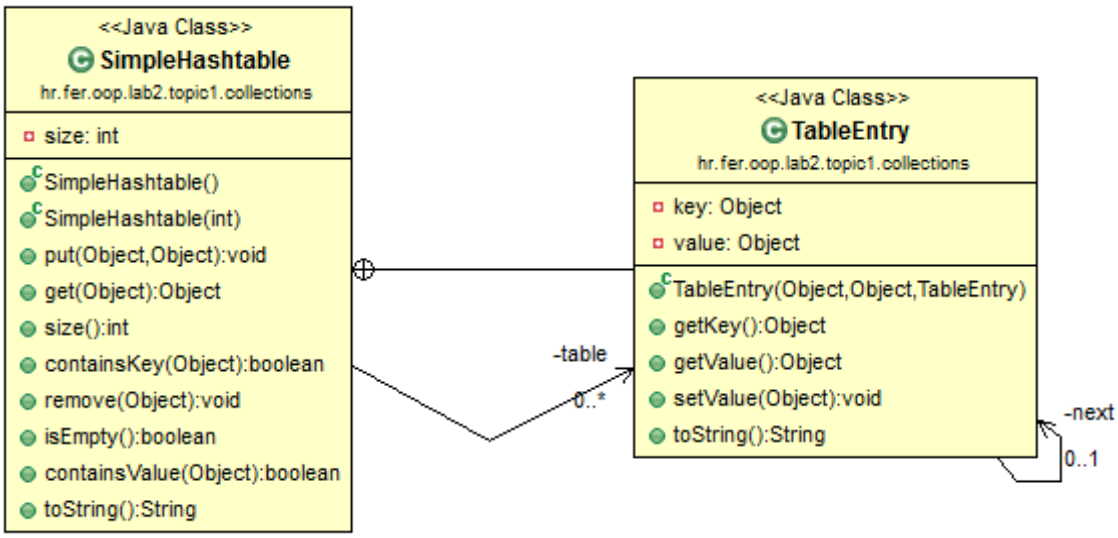
Pri tome na platformi Java 8 vrijedi:

Objekt	hashCode()	|hashCode()|	slot=|hashCode()| % 2
"Ivana"	71029095	71029095	1
"Ante"	2045822	2045822	0
"Jasna"	71344303	71344303	1
"Kristina"	-1221180583	1221180583	1

U slučaju da je u proteklih par godina došlo do izmjena u internom algoritmu koji razred String koristi za izračun hash-vrijednosti, moguće je da dobijete vrijednosti koje su drugačije od prikazanih u gornjoj tablici (a time potencijalno i drugačiju sliku). Stoga će ključevi *Ivana*, *Jasna* i *Kristina* biti u slotu 1 a ključ *Ante* u slotu 0. Razmislite odgovara li prikazana slika stvarnom stanju u memoriji ili bismo za stvarno stanje dio slike trebali drugačije nacrtati?

Dijagram razreda koji prikazuje **neparametrizirane** razrede ovog zadatka prikazan je u nastavku.

Smjestite razrede iz ovog zadatka u paket `hr.fer.zemris.java.hw04.collections`. Zanemarite paket koji je prikazan na dijagramu razreda na sljedećoj stranici.



Metode i konstruktori koje razred `SimpleHashtable` mora ponuditi navedeni su u nastavku ove upute bez posebne dokumentacije (iz imena bi moralo biti jasno što se od metode očekuje).

```
public SimpleHashtable();
public SimpleHashtable(int capacity);
public void put(K key, V value);
public V get(Object key);
public int size();
public boolean containsKey(Object key);
public boolean containsValue(Object value);
public void remove(Object key);
public boolean isEmpty();
public String toString();
```

Metoda `put` pozvana s ključem koji u tablici već postoji ažurira postojeći par novom vrijednošću; metoda ne dodaje još jedan par s istim ključem ali drugom vrijednosti. Ako se kao ključ preda `null`, metoda mora baciti `IllegalArgumentException`. Ako se zapis dodaje, onda se u odgovarajuću listu dodaje na njen kraj. Za potrebe usporedbe jesu li dva ključa ista koristite metodu `equals(other)` nad ključevima.

Metoda `get` pozvana s ključem koji u tablici ne postoji vraća `null`. Ovdje je legalno kao argument predati `null` jer takav ključ doista ne postoji. Uočite da pozivom ove metode, kada se kao rezultat dobije `null` nije moguće zaključiti je li rezultat takav iz razloga što ne postoji traženi par (ključ, vrijednosti) ili iz razloga što takav postoji, ali je vrijednost `null`.

Metoda `remove` uklanja iz tablice uređeni par sa zadanim ključem, ako takav postoji (inače ne radi ništa). Ako se kao ključ preda `null`, metoda ne radi ništa jer takav ključ doista ne postoji.

Metode `containsKey` i `containsValue` obje moraju biti u stanju primiti `null` i ispravno odreagirati, s obzirom da ključ ne može biti takav a vrijednost **može**.

Što možete zaključiti o složenosti metode `containsKey` a što o složenosti metode `containsValue` u ovako implementiranoj kolekciji (uz pretpostavku da je broj parova dodanih u tablicu dosta manji od broja slotova tablice te da funkcija sažetka radi dobro raspršenje)?

Uočite također kako su parametri metode ograničeni tipom ako su metode takve da u kolekciju dodaju podatke – tada je važna kontrola tipova. Ako su metode takve da ispituju kolekciju, metode (osim ako za to nema posebnih razloga) primaju reference općenitog tipa `Object`.

Implementirajte metodu `toString()` tako da generira popis uređenih parova koji su pohranjeni u kolekciji i koji je formatiran kako je prikazano u nastavku:

`"[key1=value1, key2=value2, key3=value3]".`

Uređeni parovi moraju biti prikazani redoslijedom koji se nalaze i tablici (od slota 0 prema dnu; u listi od prvog čvora prema zadnjem).

Problem 2.

Modificirajte prethodno razvijenu kolekciju tako da prati popunjenost. Naime, poznato je da tablice raspršenog adresiranja nude povoljne računske kompleksnosti samo ako nisu prepunjene (odnosno ako nema previše preljeva). Stoga pratite u kodu kada popunjenost tablice postane jednaka ili veća od 75% broja slotova, i u tom trenutku povećajte kapacitet tablice na dvostruki. Obratite pažnju da ćete tada iz "stare" tablice morati izvaditi sve postojeće parove i nanovo ih ubaciti u veću tablicu (jer će se promijeniti adrese u kojima ih očekujete).

Implementacijski detalj: svaka implementacija koja će ovo odraditi na način da alocira kompletan novi primjerak razreda SimpleHashTable, napuni ga podatcima i potom mu "ukrade" tablicu i odbaci taj redundantno stvoreni primjerak smatrat će se lošom implementacijom. Popis metoda koje su dane u ovoj uputi tretirajte kao javno sučelje razreda: slobodni ste dodati potreban broj privatnih pomoćnih metoda kojima ćete postići kvalitetnu organizaciju koda i izbjeći redundantno stvaranje privremenih objekata.

Dodajte metodu:

```
public void clear();
```

čija je zadaća izbrisati sve uređene parove iz kolekcije. Ova metoda ne mijenja kapacitet same tablice.

Problem 3.

Modificirajte razred `SimpleHashtable<K,V>` tako da definirate da razred implementira sučelje `Iterable<SimpleHashtable.TableEntry<K,V>>`, kako je prikazano u nastavku.

```
public class SimpleHashtable<K,V> implements Iterable<SimpleHashtable.TableEntry<K,V>> { ... }
```

Zbog ove promjene u razred ćete morati dodati metodu tvornicu koja će proizvoditi iteratore koji se mogu koristiti za obilazak po svim parovima koji su trenutno pohranjeni u tablici, i to redoslijedom kojim se nalaze u tablici ako se tablica prolazi od slota 0 prema dolje, unutar svakog slota od prvog čvora liste prema posljednjem:

```
Iterator<SimpleHashtable.TableEntry<K,V>> iterator() { ... }
```

Ideja je osigurati da možete napisati sljedeći isječak koda.

```
public class Primjer {

	public static void main(String[] args) {
		// create collection:
		SimpleHashtable<String,Integer> examMarks = new SimpleHashtable<>(2);

		// fill data:
		examMarks.put("Ivana", 2);
		examMarks.put("Ante", 2);
		examMarks.put("Jasna", 2);
		examMarks.put("Kristina", 5);
		examMarks.put("Ivana", 5); // overwrites old grade for Ivana

		for(SimpleHashtable.TableEntry<String,Integer> pair : examMarks) {
			System.out.printf("%s => %d%n", pair.getKey(), pair.getValue());
		}
	}
}
```

Ovaj kod trebao bi rezultirati sljedećim ispisom (opet napomena: ovisno o internoj implementaciji metode hashCode u razredu String, moguć je drugačiji poredak – ali svim studentima će na istoj verziji Jave biti međusobno jednak):

```
Ante => 2
Ivana => 5
Jasna => 2
Kristina => 5
```

Kod prikazan u nastavku također bi morao raditi i ispisati kartezijev produkt uređenih parova:

```
for(SimpleHashtable.TableEntry<String,Integer> pair1 : examMarks) {
	for(SimpleHashtable.TableEntry<String,Integer> pair2 : examMarks) {
		System.out.printf(
			"(%s => %d) - (%s => %d)%n",
			pair1.getKey(), pair1.getValue(),
			pair2.getKey(), pair2.getValue(),
		);
	}
}
```

Razred koji ostvaruje iterator modelirajte kao ugniježđeni razred razreda SimpleHashtable<K,V>:

```
private class IteratorImpl implements Iterator<SimpleHashtable.TableEntry<K,V>> {
	boolean hasNext() { ... }
	SimpleHashtable.TableEntry next() { ... }
	void remove() { ... }
}
```

Napisani iterator mora pri tome raditi direktno nad tablicom raspršenog adresiranja. Nije dopušteno da pri stvaranju (ili u bilo kojem trenutku kasnije) iterator prekopira sadržaj tablice u neku drugu strukturu pa radi nad njom.

Vaša implementacija Iteratora mora podržati i operaciju remove() čijim se pozivom iz tablice briše trenutni element (onaj koji je vraćen posljednjim pozivom metode next()). Uočite da je tu metodu dozvoljeno pozvati samo jednom nakon poziva metode next(). Pogledajte dokumentaciju:

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html#remove--

Obratite također pažnju da korisnik *ne mora* uopće pozivati metodu hasNext() pri iteriranju. Umjesto toga, može pozivati samo next() i čekati na NoSuchElementException koji označava da je iteriranje gotovo (takav se način koristi kao uobičajeni u Pythonu). Posljedica: nije zadaća metode hasNext() da ona računa koji je sljedeći element – pripazite na to pri pisanju koda.

Modifikacije kolekcije dok traje iteriranje

S obzirom da je efikasan algoritam iteriranja dosta teško (a ponekad i nemoguće) ostvariti ako se dozvoli da korisnik izvana modificira kolekciju, iteratore najčešće pišemo tako da svoj posao obavljaju sve dok ne uoče da je kolekcija izvana modificirana; kada to utvrde, svi pozivi metoda iteratora bacaju iznimku ConcurrentModificationException čime odbijaju daljnje iteriranje.

U sljedećem primjeru, iz kolekcije se uklanja ocjena za Ivanu na korektan način (nema iznimke).

```
Iterator<SimpleHashtable.TableEntry<String,Integer>> iter = examMarks.iterator();
while(iter.hasNext()) {
	SimpleHashtable.TableEntry<String,Integer> pair = iter.next();
	if(pair.getKey().equals("Ivana")) {
		iter.remove();  // sam iterator kontrolirano uklanja trenutni element
	}
}
```

Sljedeći kod bacio bi IllegalStateException jer se uklanjanje poziva više od jednom za trenutni par nad kojim je iterator (to bi bacio drugi poziv metode remove):

```
Iterator<SimpleHashtable.TableEntry<String,Integer>> iter = examMarks.iterator();
while(iter.hasNext()) {
	SimpleHashtable.TableEntry<String,Integer> pair = iter.next();
	if(pair.getKey().equals("Ivana")) {
		iter.remove();
		iter.remove();
	}
}
```

Sljedeći kod bacio bi ConcurrentModificationException jer se uklanjanje poziva "izvana" (direktno nad kolekcijom a ne kroz iterator koji to može obaviti kontrolirano i ažurirati svoje interne podatke). Iznimku bi bacila metoda hasNext jer je ona prva koja se u prikazanom primjeru poziva nakon brisanja.

```
Iterator<SimpleHashtable.TableEntry<String,Integer>> iter = examMarks.iterator();
while(iter.hasNext()) {
	SimpleHashtable.TableEntry<String,Integer> pair = iter.next();
	if(pair.getKey().equals("Ivana")) {
		examMarks.remove("Ivana");
	}
}
```

Kako implementirati ovakvo ponašanje? Evo ideje. Opremite razred SimpleHashtable<K,V> još jednom privatnom članskom varijablom: modificationCount. Svaka metoda razreda SimpleHashtable<K,V> koja na bilo koji način strukturno mijenja internu podatkovnu strukturu kolekcije (npr. dodavanje novog para rezultira promjenom ulančane liste u odgovarajućem slotu ili stvaranjem nove glave liste ako je slot bio prazan, promjena veličine tablice) treba ovaj brojač povećati za jedan. Ako ne dolazi do promjene interne podatkovne strukture, primjerice, situacija gdje korisnik poziva metodu put za ključ koji već postoji pa se tada samo ažurira pridružena vrijednost, tada se brojač ne ažurira. Alternativno, mogli bismo biti puno stroži pa reći da ćemo brojač ažurirati pri bilo kakvoj promjeni nad kolekcijom; no to nećemo raditi u ovoj zadaći.

Svaki iterator pri stvaranju mora zapamtiti koju je vrijednost imao taj brojač u trenutku stvaranja iteratora. Pri svakom pozivu *bilo koje od metoda iteratora*, iterator *najprije* uspoređuje zapamćenu vrijednost brojača s trenutnom vrijednosti brojača u kolekciji – kako je implementacija iteratora privatni ugniježđeni razred u razredu koji modelira samu kolekciju, njegov kod ima pristup toj članskoj varijabli kolekcije; ako utvrdi da se vrijednosti ne podudaraju, baca iznimku. Prilikom implementacije metode remove u iteratoru pripazite da ažurirate zapamćenu vrijednost (čak i kada iterator radi brisanje, on mora povećati modificationCount jer je moguće da neki drugi iterator paralelno obilazi elemente iste kolekcije – tada taj treba biti u stanju otkriti da je došlo do modifikacije; no naš iterator sebi također mora ažurirati očekivanu vrijednost brojača kako bi dalje mogao nastaviti s radom). S obzirom da ćete brisanje elemenata u kolekciji raditi na dva načina (bilo izravnim pozivom metode remove same kolekcije ili pozivom metode remove iteratora), razmislite kako ćete napisati taj kod da nemate redundantnog koda.

Sljedeći kod trebao bi ispisati sve parove i po završetku ostaviti kolekciju praznom.

```
Iterator<SimpleHashtable.TableEntry<String,Integer>> iter = examMarks.iterator();
while(iter.hasNext()) {
        SimpleHashtable.TableEntry<String,Integer> pair = iter.next();
        System.out.printf("%s => %d%n", pair.getKey(), pair.getValue());
        iter.remove();
}
System.out.printf("Veličina: %d%n", examMarks.size());
```

Problem 4.

Please read the whole problem description before you start coding – there are implementation details later in the text.

Write a simple database emulator. Put the implementation classes in package `hr.fer.zemris.java.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`. In order to achieve this, 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). 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 `SimpleHashtable` class from problems 1 to 3.

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

```
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 perform 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 `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);
```

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

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 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 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 can queries of this form as 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 direct one, method must throw `IllegalStateException`.

`List<ConditionalExpression> getQuery();`
=> For all queries, 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 please do not lose time on optimization. For example, query such `jmbag="1" and jmbag="2"` is obviously false always, 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.zemris.java.tecaj.hw04.db.StudentRecord
hr.fer.zemris.java.tecaj.hw04.db.IComparisonOperator
hr.fer.zemris.java.tecaj.hw04.db.ComparisonOperators
hr.fer.zemris.java.tecaj.hw04.db.IFieldValueGetter
hr.fer.zemris.java.tecaj.hw04.db.FieldValueGetters
hr.fer.zemris.java.tecaj.hw04.db.ConditionalExpression
hr.fer.zemris.java.tecaj.hw04.db.QueryParser
hr.fer.zemris.java.tecaj.hw04.db.StudentDatabase
hr.fer.zemris.java.tecaj.hw04.db.IFilter
hr.fer.zemris.java.tecaj.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!
5. Write and test `IfieldValueGetter`/`FieldValueGetters`.
6. Write and test `ConditionalExpression`.
7. Write and test `QueryParser`.
8. Write and test `QueryFilter`.

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). Additionally, for this homework you can not use any of Java Collection Framework classes which represent collections or its derivatives for problems 1 to 3; in problem 4 you can use java collections (i.e. lists, but observe that implementation of database indexing is explicitly defined to use your implementation of SimpleHashtable). Document your code!

If you need any help, I have reserved Monday and Tuesday at 1 PM a slot for consultations. Feel free to drop by my office.

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 tests for problem 4 (see "Recommended order of writing"). You are encouraged to write tests for problems 1-3.

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. Deadline is March 30th 2017. at 07:00 AM.