3rd homework assignment; OPRPP1

Napravite prazan Maven projekt: u Eclipsovom workspace direktoriju napravite direktorij `hw03-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.oprpp1.jmbag0000000000:hw03-0000000000` (zamijenite nule Vašim JMBAG-om) i dodajte ovisnost prema `junit5`. Importajte projekt u Eclipse. Sada možete nastaviti s rješavanjem zadataka.

Problem 1.

Iz Vaše druge domaće zadaće uzmite paket `hr.fer.oprpp1.custom.collections` i prekopirajte sve relevantne razrede/sučelja ovdje (`Collection`, `List`, `ArrayIndexedCollection`, `LinkedListIndexedCollection`, `ObjectStack`, `Processor` te ostala povezana sučelja). Provedite parametrizaciju Vaše implementacije kolekcija, tako da možete imati primjerice `Collection<String>`, `List<Integer>`, `ElementsGetter<Double>`, `Processor<String>`, `Tester<String>`, `ObjectStack<List<String>>` i slično. Ako za prikladnu vrijednost parametra kod nekih od metoda niste sigurni, slobodno pogledajte izvorni kod od Java Collection Framework (ili barem javadoc analognih razreda i sučelja ovima koje smo napravili). Kada se pitate treba li nam na nekom mjestu uopće parametar, vodite se sljedećim: kod metoda koje mijenjaju kolekciju umetanjem objekata, želimo da već i sam kompajler može provjeriti legalnost; kod metoda koje nešto ispituju (poput metode `contains`), želimo maksimalnu fleksibilnost (ako sam kolekcija Stringova i ako me netko pita sadržim li boeing747 koji je Avion, upit želimo smatrati legalnim i očekivani odgovor je negativan). Razmislite koje su posljedice ovoga.

Problem 2.

U paket `hr.fer.oprpp1.custom.collections` dodat ćemo još jedan razred: `Dictionary`. Radi se o razredu koji pod zadanim ključem pamti predanu vrijednost (u svijetu Jave, ovakve strukture poznate su kao mape; u PHP-u i JavaScriptu kao asocijativna polja; u Pythonu i C# kao riječnici). U ovom zadatku, s obzirom na mali broj zapisa koje želimo pohranjivati, naglasak neće biti na učinkovitosti već na jednostavnosti. Stoga ćemo modelirati parametrizirani razred `Dictionary` kao *adapter* (vidi 2. domaću zadaću za pojam) oko `ArrayIndexedCollection`, pri čemu razred definira dva parametra: K i V (prvi je tip ključa, drugi tip vrijednosti). Jedan zapis (tj. uređeni par) modelirajte privatnim ugniježđenim razredom koji će omogućiti pamćenje dva podatka: `key` i `value`. Ključ pri tome ne smije biti `null`; vrijednost smije. Razred `Dictionary` mora korisnicima ponuditi sljedeće metode.

```
boolean isEmpty();
int size();
void clear();
V put(K key, V value);          // "gazi" eventualni postojeći zapis
V get(Object key);              // ako ne postoji pripadni value, vraća null
V remove(K key);                // uklanja zapis za ključ (ako postoji)
```

Važno: ako u riječniku već postoji zapis za neki ključ, tada će poziv `put` s tim istim ključem zamijeniti upisanu vrijednost onom koja je predana kao drugi argument od `put` (povratna vrijednost je ta stara vrijednost, ili `null` ako prethodno nije bilo zapisa).

Metoda `remove` uklanja zapis (čitav zapis; ne postavlja samo `value` na `null`) sa zadanim ključem (ako takav zapis postoji) i vraća vrijednost koja je bila upisana u riječniku za taj ključ (ako je postojala); inače vraća `null`.

Napišite junit testove za Vašu implementaciju ovog zadatka!

Problem 3.

Napravite paket `hr.fer.oprpp1.math` i u njega smjestite razred `Vector2D`. Razred modelira 2D vektor čije su komponente realni brojevi `x` i `y`. Razred treba ponuditi sljedeće konstruktore/metode.

```
public Vector2D(double x, double y);
public double getX();
public double getY();
public void add(Vector2D offset);
public Vector2D added(Vector2D offset);
public void rotate(double angle);
public Vector2D rotated(double angle);
public void scale(double scaler);
public Vector2D scaled(double scaler);
public Vector2D copy();                        // returns new copy of vector
```

Pri tome operacije `add`/`rotate`/`scale` direktno modificiraju trenutni vektor dok metode `added`/`rotated`/`scaled` vraćaju novi vektor koji odgovara rezultatu primjene zadane operacije nad trenutnim vektorom (pri čemu se trenutni vektor ne mijenja).

Napišite junit testove za sve metode!

Problem 4a.

U ovom zadatku razmatramo malo ozbiljniju implementaciju mape. 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.

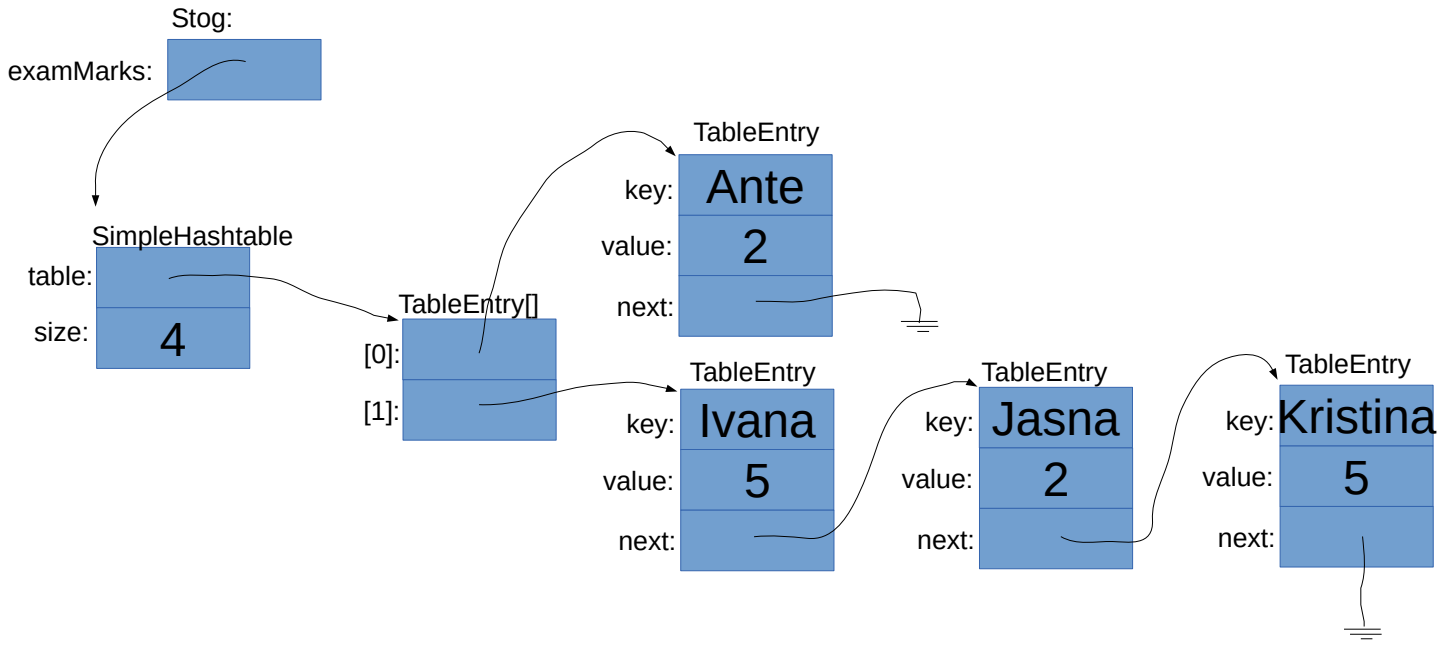
```
// create collection:
SimpleHashtable<String, Integer> examMarks = new SimpleHashtable<>(2);

// 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.



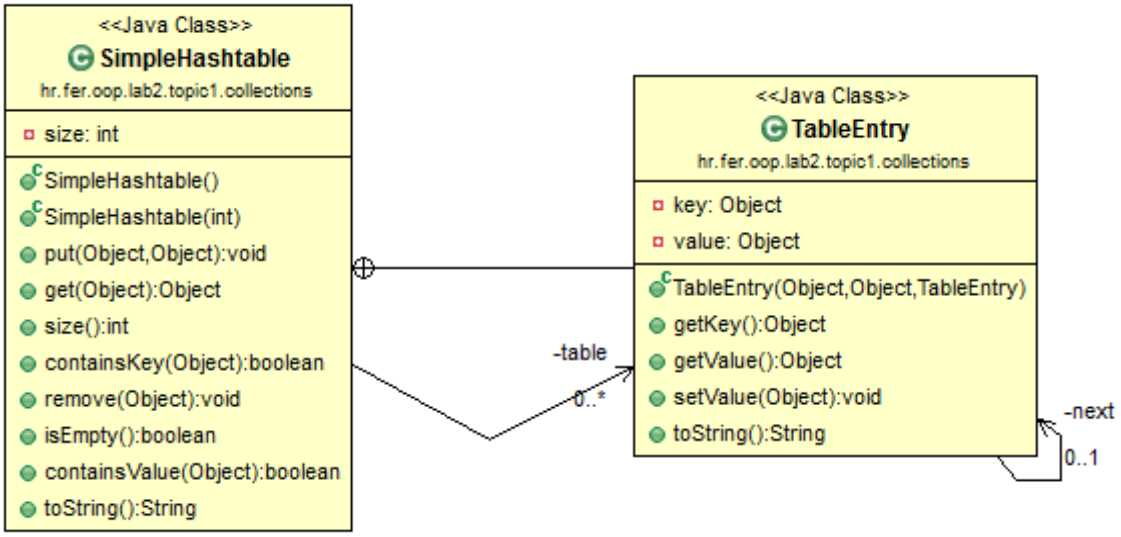
Primijetite na slici: veličina kolekcije (`size`) i kapacitet internog polja (veličina tablice) su različite stvari. Na platformi Java 8 vrijedi (na drugim verzijama moguće su druge vrijednosti!):

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). U slučaju da su hash-vrijednosti kao u prikazanoj tablici, zapisi za ključeve *Ivana*, *Jasna* i *Kristina* bit će u slotu 1, a zapis za ključ *Ante* u slotu 0. Razmislite odgovara li prikazana slika stvarnom stanju u memoriji ili bismo za stvarno stanje dio slike trebali još malo drugačije nacrtati?

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

Smjestite razrede iz ovog zadatka u paket `hr.fer.oprpp1.custom.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 V put(K key, V value);
public V get(Object key);
public int size();
public boolean containsKey(Object key);
public boolean containsValue(Object value);
public V remove(Object key);
public boolean isEmpty();
public String toString();
public TableEntry<K,V>[] toArray();
```

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 `NullPointerException`. Ako se zapis dodaje, onda se u odgovarajuću listu dodaje na njen kraj, te pozivatelju vraća `null`; ako se gazi neka prethodna vrijednost, onda je metoda vraća pozivatelju. 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 i vraća pozivatelju tu vrijednost (inače ne radi ništa i vraća `null`). 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).

Metoda `toArray` vraća polje referenci na pohranjene zapise; to polje ima onoliko elemenata koliki je trenutni `size` kolekcije (dakle, to nije isto što i interna tablica). Elemente je u to polje potrebno pohraniti redoslijedom kojim se nalaze u internoj strukturi mape.

Problem 4b.

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 u trenutku prije no što pokušate dodati novi zapis u tablicu pratite u kodu kada popunjenost tablice (definirana kao omjer sadržaja varijable size i veličine internog polja referenci) postane jednaka ili veća od 75% broja slotova; ako je to ispunjeno, povećajte kapacitet tablice na dvostruki (pa tek tada nastavite s operacijom dodavanja novog zapisa). Obratite pažnju da ćete tada iz "stare" tablice morati izvaditi sve postojeće parove i nanovo ih ubaciti u veću tablicu (jer će se promijeniti slotovi 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. Primjerice, ovdje bi Vam mogla pomoći metoda `toArray`*.*

Dodajte metodu:

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

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

Problem 4c.

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());
```

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) except the code which you are required to copy from your previous homework. Additionally, for this homework you can not use any of Java Collection Framework classes which represent collections or its derivatives for problems 1 to 4. 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 tests for problems 2 and 3.
You are encouraged to write tests for other problems.

When your homework is done, pack it in zip archive with name `hw03-0000000000.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.