3. homework assignment; JAVA, Academic year 2018/2019; FER

Napravite prazan Maven projekt, kao u 1. zadaći: u Eclipsovom workspace direktoriju napravite direktorij `hw03-0000000000` (zamijenite nule Vašim JMBAG-om) te u njemu oformite Mavenov projekt `hr.fer.zemris.java.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.zemris.java.custom.collections` i prekopirajte ga ovdje. Konkretno, trebate razrede `Collection`, `ArrayIndexedCollection`, `LinkedListIndexedCollection` i `Processor`. U ovom zadatku modificirat ćemo napisano rješenje i malo ga obogatiti. U isti paket stavljat ćete i druga nova sučelja koja slijede kroz zadatak. Demonstracijske programe možete stavljati u zaseban paket `demo`.

Podzadatak 1.

Razred `Collection` pretvorite u sučelje. Sve prethodno "prazne" metode ili metode s pogrešnom implementacijom pretvorite u apstraktne. Ponudite defaultnu implementaciju za metodu `addAll(Collection other)` te metodu `isEmpty()`.

Zbog ovoga ćete morati popraviti i razrede `ArrayIndexedCollection` te `LinkedListIndexedCollection` jer se inače kod neće moći prevesti. Sada imate priliku i popraviti javadoc apstraktnih metoda specificiranih u sučelju s obzirom da iste više ne rade besmislene stvari.

Također, pretvorite razred `Processor` u sučelje s jednom apstraktnom metodom te popravite druga mjesta u kodu koja se zbog te izmjene sada više ne mogu prevesti.

Krenite na podzadatak 2 nakon što ste kod doveli u stanje da se može ponovno prevesti.

Podzadatak 2.

Jedan od problema kolekcije modelirane prethodno razredom a sada sučeljem `Collection` jest nemogućnost dohvata elemenata koji su u toj kolekciji, osim metodom `toArray` koja ih sve sprema u novo polje, ili pak procesiranjem uporabom metode `forEach`.

Stoga ćemo sada osmisliti novu vrstu objekata: `ElementsGetter` čija je zadaća korisniku vraćati element po element, na korisnikov zahtjev. Svaka kolekcija će nam ponuditi metodu `createElementsGetter(): ElementsGetter` za stvaranje takvih objekata, a sami objekti s korisnikom će komunicirati kroz dvije metode: `hasNextElement(): boolean`, te `getNextElement(): Object`.

Evo ideje kako ovo želimo koristiti.

```
public static void main(String[] args) {
	Collection col = stvoriPraznuKolekciju(); // npr. new ArrayIndexedCollection();
	col.add("Ivo");
	col.add("Ana");
	col.add("Jasna");

	ElementsGetter getter = col.createElementsGetter();

	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Jedan element: " + getter.getNextElement());

	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Jedan element: " + getter.getNextElement());

	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Jedan element: " + getter.getNextElement());

	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Jedan element: " + getter.getNextElement());

}
```

Kad nam kolekcija vrati novi objekt koji koristimo za dohvat njezinih elemenata, taj objekt "vidi" da kolekcija ima tri elementa i da nam on još nije vratio niti jedan. Stoga će nam objekt pri prvom pozivu metode `hasNextElement()` vratiti `true`, a na prvi poziv metode `getNextElement()` vratit će nam jedan od tri elementa koji su u kolekciji (primjerice: Ivo). Na sljedeći poziv `hasNextElement()` objekt će nam opet vratiti `true` (jer kolekcija ima tri elementa a on nam je dao tek jedan), i na poziv metode `getNextElement()` vratit će nam sljedeći od tri elementa koji su u kolekciji (primjerice: Ana). Na sljedeći poziv `hasNextElement()` objekt će nam opet vratiti `true` (jer kolekcija ima tri elementa a on nam je dao tek dva), i na poziv metode `getNextElement()` vratit će nam sljedeći od tri elementa koji su u kolekciji (primjerice: Jasna).

Konačno, na sljedeći poziv `hasNextElement()` objekt će nam vratiti `false` (jer kolekcija ima tri elementa i on nam je isporučio sva tri). Potom na poziv metode `getNextElement()` objekt će izazvati `NoSuchElementException` jer više nema neisporučenih elemenata, a mi smo tražili isporuku sljedećeg. Pazite: ovime smo definirali i očekivano ponašanje ove metode; nemojte zaboraviti na to prilikom implementacije.

Evo još par naputaka vezano uz očekivano ponašanje objekata koji su `ElementsGetter`i. Pozivi metode `hasNextElement()` su idempotentni; drugim riječima – status ima li još elemenata za isporuku može se modificirati tek nakon što se neki element isporuči. Evo primjera:

```
public static void main(String[] args) {
	Collection col = stvoriPraznuKolekciju(); // npr. new ArrayIndexedCollection();
	col.add("Ivo");
	col.add("Ana");
	col.add("Jasna");

	ElementsGetter getter = col.createElementsGetter();

	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
	System.out.println("Jedan element: " + getter.getNextElement());

	System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
```

```
        System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
        System.out.println("Jedan element: " + getter.getNextElement());

        System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
        System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
        System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
        System.out.println("Jedan element: " + getter.getNextElement());

        System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
        System.out.println("Ima nepredanih elemenata: " + getter.hasNextElement());
}
```

Program će 5 puta ispisati `true`, zatim dohvaćamo prvi element, potom će dva puta ispisati `true`, pa dohvaćamo drugi element, pa će tri puta ispisati `true`, pa dohvaćamo treći element, pa će dva puta ispisati `false`.

Također, korisnik nije dužan uopće ispitivati ima li još elemenata. Evo primjera.

```
public static void main(String[] args) {
        Collection col = stvoriPraznuKolekciju(); // npr. new ArrayIndexedCollection();
        col.add("Ivo");
        col.add("Ana");
        col.add("Jasna");

        ElementsGetter getter = col.createElementsGetter();

        System.out.println("Jedan element: " + getter.getNextElement());
        System.out.println("Jedan element: " + getter.getNextElement());
        System.out.println("Jedan element: " + getter.getNextElement());
        System.out.println("Jedan element: " + getter.getNextElement());

}
```

Program će ispisati sva tri elementa i potom baciti iznimku.

Nadalje, svi `ElementsGetter`i su međusobno nezavisni i kolekcija može imati više takvih objekata koji nad njom rade istovremeno. Primjerice:

```
public static void main(String[] args) {
        Collection col = stvoriPraznuKolekciju(); // npr. new ArrayIndexedCollection();
        col.add("Ivo");
        col.add("Ana");
        col.add("Jasna");

        ElementsGetter getter1 = col.createElementsGetter();
        ElementsGetter getter2 = col.createElementsGetter();

        System.out.println("Jedan element: " + getter1.getNextElement());
        System.out.println("Jedan element: " + getter1.getNextElement());
        System.out.println("Jedan element: " + getter2.getNextElement());
        System.out.println("Jedan element: " + getter1.getNextElement());
        System.out.println("Jedan element: " + getter2.getNextElement());

}
```

Uz pretpostavku da objekti elemente dohvaćaju redoslijedom Ivo, Ana, Jasna, prethodni kod bi ispisao Ivo, Ana, Ivo, Jasna, Ana.

Evo sada konkretnog naputka što trebate napraviti.

Modificirajte sučelje `Collection` dodavanjem apstraktne metode `createElementsGetter()`. U svakom od razreda `ArrayIndexedCollection` i `LinkedListIndexedCollection` napišite privatni statički razred koji predstavlja implementaciju `ElementsGetter`a koji zna dohvaćati i vraćati elemente na opisani način. Važno: taj razred ne smije stvarati kopiju elemenata kolekcije (ili primjerice reći kolekciji da elemente izdumpa u polje, pa trčati po tom polju); umjesto toga, `ElementsGetter` treba biti implementiran tako da izravno koristi internu strukturu podataka same kolekcije (primjerice, kod `LinkedListIndexedCollection`, objekt bi trebao pamtiti referencu na čvor liste iz kojeg treba vratiti sljedeći element). Pretpostavite za sada da će korisnik biti pristojan, pa neće dodavati/brisati elemente iz kolekcije ako istovremeno koristi neki `ElementsGetter`.

Dodajte sada u razrede `ArrayIndexedCollection` i `LinkedListIndexedCollection` implementaciju metode `createElementsGetter()` tako da na poziv stvara i vraća novi primjerak privatnog statičkog razreda koji ste upravo napisati. Dodatni ugovori koji vrijede su sljedeći. Razred `ArrayIndexedCollection` stvara objekte tipa `ElementsGetter` koji objekte vraćaju kretanjem po internom polju referenci od pozicije nula na dalje (što bi ujedno odgovaralo i redoslijedu dodavanja metodom `add`). Razred `LinkedListIndexedCollection` stvara objekte tipa `ElementsGetter` koji objekte vraćaju kretanjem po internoj ulančanoj listi od prvog čvora na dalje (što bi također odgovaralo i redoslijedu dodavanja metodom `add`). Provjerite sada sljedeći kod.

```
public static void main(String[] args) {
	Collection col1 = new ArrayIndexedCollection();
	Collection col2 = new ArrayIndexedCollection();
	col1.add("Ivo");
	col1.add("Ana");
	col1.add("Jasna");
	col2.add("Jasmina");
	col2.add("Štefanija");
	col2.add("Karmela");

	ElementsGetter getter1 = col1.createElementsGetter();
	ElementsGetter getter2 = col1.createElementsGetter();
	ElementsGetter getter3 = col2.createElementsGetter();

	System.out.println("Jedan element: " + getter1.getNextElement());
	System.out.println("Jedan element: " + getter1.getNextElement());
	System.out.println("Jedan element: " + getter2.getNextElement());
	System.out.println("Jedan element: " + getter3.getNextElement());
	System.out.println("Jedan element: " + getter3.getNextElement());

}
```

Ispis treba biti: Ivo, Ana, Ivo, Jasmina, Štefanija. Ne idite dalje dok ovo niste riješili.

Podzadatak 3.

Sada idemo osigurati da nas nije briga ponaša li se korisnik pristojno. Konkretno, ako naš `ElementsGetter` prati reference po ulančanoj listi, i korisnik promijeni strukturu te liste (brisanjem čvorova ili umetanjem čvorova), tada naš `ElementsGetter` treba toga postati svjestan, i na poziv bilo koje svoje metode treba baciti iznimku `ConcurrentModificationException`. To ćemo postići na sljedeći način.

Modificirajte razrede `ArrayIndexedCollection` i `LinkedListIndexedCollection` tako da deklariraju privatnu člansku varijablu `modificationCount` tipa `long` i početne vrijednosti nula. Svaki puta kada korisnik nad kolekcijom pozove metodu koja radi strukturnu modifikaciju (kod polja to znači realocira polje

ili pomiče elemente lijevo ili desno, kod liste to znači dodaje ili briše čvorove), kolekcija treba povećati sadržaj varijable `modificationCount` za jedan. Privatni statički razredi koji implementiraju `ElementsGetter` trebaju i sami deklarirati dodatnu privatnu člansku varijablu `savedModificationCount` i u konstruktoru njezin sadržaj trebaju postaviti na sadržaj varijable `modificationCount` kolekcije čije elemente trebaju vratiti. I potom, modificirajte metode koje ti privatni razredi imaju zbog toga što implementiraju sučelje `ElementsGetter` tako da prije svega usporede aktualni sadržaj varijable `modificationCount` kolekcije nad kojom operiraju i sadržaj varijable `savedModificationCount` koji su zapamtili prilikom njihova stvaranja: ako su ti sadržaji različiti, znači da je kolekcija strukturno mijenjana i odmah treba baciti iznimku `ConcurrentModificationException`. Provjerite sada sljedeći kod:

```
public static void main(String[] args) {
	Collection col = new ArrayIndexedCollection();
	col.add("Ivo");
	col.add("Ana");
	col.add("Jasna");

	ElementsGetter getter = col.createElementsGetter();

	System.out.println("Jedan element: " + getter.getNextElement());
	System.out.println("Jedan element: " + getter.getNextElement());

	col.clear();

	System.out.println("Jedan element: " + getter.getNextElement());

}
```

Program treba ispisati Ivo, Ana pa se raspasti s iznimkom `ConcurrentModificationException`. Provjerite isto ponašanje i za `LinkedListIndexedCollection`.

Podzadatak 4.

Dodajte sada u sučelje `ElementsGetter` defaultnu implementaciju metode:

```
void processRemaining(Processor p);
```

koja će nad svim preostalim elementima kolekcije pozvati zadani procesor. Primjer:

```
public static void main(String[] args) {
	Collection col = new ArrayIndexedCollection();
	col.add("Ivo");
	col.add("Ana");
	col.add("Jasna");

	ElementsGetter getter = col.createElementsGetter();
	getter.getNextElement();

	getter.forEachRemaining(System.out::println);
}
```

Ovaj kod na ekran će ispisati Ana pa Jasna.

Podzadatak 5.

Sučeljem `Tester` modelirat ćemo objekte koji prime neki objekt te ispitaju je li taj objekt prihvatljiv ili nije (vraćaju `true` ako ga prihvaćaju, `false` ako ga ne prihvaćaju). Sučelje treba imati samo jednu apstraktnu

metodu:

```
boolean test(Object obj);
```

Sada možemo napisati jednu implementaciju testera koji provjeravaju jesu li objekti parni integeri (stavite taj razred u paket `demo` gdje pišete i ostale primjere):

```
class EvenIntegerTester implements Tester {
  public boolean test(Object obj) {
    if(!(obj instanceof Integer)) return false;
    Integer i = (Integer)obj;
    return i % 2 == 0;
  }
}

Tester t = new EvenIntegerTester();
System.out.println(t.test("Ivo"));
System.out.println(t.test(22));
System.out.println(t.test(3));
```

Program bi trebao ispisati: `false` pa `true` pa `false`.

Modificirajte sada sučelje `Collection`, tako da dodate defaultnu metodu:

```
void addAllSatisfying(Collection col, Tester tester);
```

Metoda treba dohvaćati redom sve elemente iz predane kolekcije `col` (koristite njezin `ElementsGetter`), te u trenutnu kolekciju treba na kraj dodati sve elemente koje predani `tester` prihvati. Time će sljedeći kod postati moguć:

```
public static void main(String[] args) {
      Collection col1 = new LinkedListIndexedCollection();
      Collection col2 = new ArrayIndexedCollection();

      col1.add(2);
      col1.add(3);
      col1.add(4);
      col1.add(5);
      col1.add(6);

      col2.add(12);
      col2.addAllSatisfying(col1, new EvenIntegerTester());

      col2.forEach(System.out::println);
}
```

Ako ste sve dobro implementirali, program treba ispisati brojeve: 12, 2, 4, 6.

Podzadatak 6: završna čišćenja

Sada kada naše kolekcije znaju stvarati objekte tipa `ElementsGetter`, idemo malo počistiti kod. Uklonite iz razreda `ArrayIndexedCollection` i `LinkedListIndexedCollection` implementacije metode `forEach` (obrišite ih); dodajte u razred `Collection` defaultnu implementaciju te metode koja se oslanja na `ElementsGetter`.

Definirajte novo sučelje `List` koje nasljeđuje sučelje `Collection` i koje definira sljedeće metode:

```
Object get(int index);
void insert(Object value, int position);
int indexOf(Object value);
void remove(int index);
```

Sada preradite razrede `ArrayIndexedCollection` i `LinkedListIndexedCollection` tako da implementiraju sučelje `List` a ne izravno `Collection` pa time možemo pisati sljedeći kod:

```
List col1 = new ArrayIndexedCollection();
List col2 = new LinkedListIndexedCollection();
col1.add("Ivana");
col2.add("Jasna");

Collection col3 = col1;
Collection col4 = col2;

col1.get(0);
col2.get(0);
col3.get(0); // neće se prevesti! Razumijete li zašto?
col4.get(0); // neće se prevesti! Razumijete li zašto?

col1.forEach(System.out::println); // Ivana
col2.forEach(System.out::println); // Jasna
col3.forEach(System.out::println); // Ivana
col4.forEach(System.out::println); // Jasna
```

Problem 2.

U okviru ovog zadatka pripremamo se za izradu jednostavnog jezičnog procesora. Jezični procesor uobičajeno se sastoji od nekoliko dijelova. Prvi dio je podsustav za izradu leksičke analize. Ulaz ovog podsustava je izvorni tekst programa (dokumenta ili što se već obrađuje) a izlaz je niz *tokena*. Token je leksička jedinica koja grupira jedan ili više uzastopnih znakova iz ulaznog teksta. Primjerice, ako je ulaz izvorni tekst razreda u Javi, tokeni bi bile ključne riječi, identifikatori, cijeli brojevi, decimalni brojevi, simboli, itd. Za tokene pamtimo kojeg su tipa te koju vrijednost predstavljaju. Primjerice, token čiji je tip "ključna riječ" može imati kao vrijednost string "for" ili pak "do" ili "while" ili "new" ili … Token čiji je tip "cijeli broj" može imati kao vrijednost podatak tipa long vrijednosti 358 i slično (pogledajte u knjizi u poglavlju 18 sliku 18.1 – samo nju, ne trebate čitati to poglavlje u ovom trenutku).

Podsustav koji obavlja sintaksnu analizu konzumira niz tokena koje generira podsustav za leksičku analizu i gradi primjerice sintaksno stablo. Takav će podsustav zaključiti da, ako vidi slijed tokena koji je identifikator (tj. niz slova) pa tokena koji je simbol "=", da se radi o pridjeljivanju vrijednosti varijabli (kao u tekstu "tmp=7", prvi token je tipa identifikator i čuva vrijednost "tmp" a drugi je tipa simbol i čuva vrijednost "="), a ako vidi slijed tokena koji je identifikator pa tokena koji je simbol "(", da se radi o pozivu metode (kao u tekstu "print(x)")

Leksički analizator uobičajeno se izvodi kao *lijeni*: grupiranje znakova odnosno ekstrakciju svakog sljedećeg tokena radi tek kada ga se to eksplicitno zatraži pozivom odgovarajuće metode za dohvat sljedećeg tokena.

Sve razrede i enumeracije u ovom zadatku smjestite u paket `hr.fer.zemris.java.hw03.prob1`.
U okviru ovog zadatka potrebno je napraviti jednostavni leksički analizator (razred `Lexer`).

```
public class Lexer {
	private char[] data;	// ulazni tekst
	private Token token;	// trenutni token
	private int currentIndex; // indeks prvog neobrađenog znaka

	// konstruktor prima ulazni tekst koji se tokenizira
	public Lexer(String text) { ... }

	// generira i vraća sljedeći token
	// baca LexerException ako dođe do pogreške
	public Token nextToken() { ... }

	// vraća zadnji generirani token; može se pozivati
	// više puta; ne pokreće generiranje sljedećeg tokena
	public Token getToken() {...}
}
```

Kostur razreda `Token` prikazan je u nastavku.

```
public class Token {
	...

	public Token(TokenType type, Object value) {...}

	public Object getValue() {...}

	public TokenType getType() {...}
}
```

`TokenType` je enumeracija čije su moguće vrijednosti `EOF`, `WORD`, `NUMBER`, `SYMBOL`. `LexerException` je iznimka koja je izvedena iz razreda `RuntimeException`. Vaš je zadatak napisati sve spomenute razrede / enumeracije.

Pravila kojih se lexer pridržava su sljedeća. Tekst se sastoji od niza riječi, brojeva te simbola. Riječ je svaki niz od jednog ili više znakova nad kojima metoda `Character.`*`isLetter`* vraća `true`. Broj je svaki niz od jedne ili više znamenaka, a koji je prikaziv u tipu `Long`. Zabranjeno je u tekstu imati prikazan broj koji ne bi bio prikaziv tipom `Long` (u tom slučaju lexer mora baciti iznimku: ulaz ne valja!). Simbol je svaki pojedinačni znak koji se dobije kada se iz ulaznog teksta uklone sve riječi i brojevi te sve praznine (`'\r'`, `'\n'`, `'\t'`, `' '`). Praznine ne generiraju nikakve tokene (zanemaruju se). Vrijednosti koje su riječi pohranjuju se kao primjerci razreda `String`, brojevi kao primjerci razreda `Long` a simboli kao primjerci razredi `Character`. Token tipa `EOF` generira se kao posljednji token u obradi, nakon što lexer grupira sve znakove iz ulaza i bude ponovno pozvad da obavi daljnje grupiranje. U slučaju da korisnik nakon što lexer vrati token tipa `EOF` opet zatraži generiranje sljedećeg tokena, lexer treba baciti iznimku. U knjizi je primjer jednog lexera dan u primjeru 18.5 (razred VLangTokenizer, stranica 463; *opet napomena*: možete pogledati ideju realizacije).

Pretpostavite da je u datoteci zapisan sljedeći tekst.

```
Ovo je 123ica, ab57.
Kraj
```

Za potrebe testiranja, ovaj tekst možemo pohraniti kao `String` varijablu u Javi:

```
String ulaz = "Ovo je 123ica, ab57.\nKraj";
```

Očekivali bismo da uzastopnim pozivima metode `nextToken` nad lexerom koji je inicijaliziran prikazanim tekstom dobijemo sljedeći niz tokena:

```
(WORD, Ovo)
(WORD, je)
(NUMBER, 123)
(WORD, ica)
(SYMBOL, ,)
(WORD, ab)
(NUMBER, 57)
(SYMBOL, .)
(WORD, Kraj)
(EOF, null)
```

U posljednjem prikazanom tokenu vrijednost je `null`-referenca.

Dopušta se uporaba mehanizma escapeanja: ako se u tekstu ispred znaka koji je znamenka nađe znak `\`, ta se znamenka tretira kao slovo. Posljedica je da tako escapeano slovo može biti dio riječi. Evo primjera i ispod njega očekivane tokenizacije. Znak `\` ispred samog sebe također predstavlja ispravnu escape sekvencu koja označava znak `\` tretiran kao slovo. Niti jedna escape sekvenca osim opisanih nije valjana i za njih lexer treba baciti iznimku.

```
\1\2 ab\\\2c\3\4d
```

```
(WORD, 12)
(WORD, ab\2c34d)
(EOF, null)
```

Primjetite da bi, zbog pravila escapeanja string-konstanti u Javi, prethodni ulaz u izvornom kodu izgledao:

```
String ulaz = "\\1\\2 ab\\\\\\2c\\3\\4d";
```

Naime, u String literalu koji pišete u Java izvornom kodu znak \ označava escape-sekvencu za Javin kompajler. Želite li u Javi definirati string koji ima *jedan* znak \, pišete `"\\"`. Vrijedi `"\\".length()=1`.

Da biste riješili zadatak, napravite sljedeće.

1. Napišite enumeraciju `TokenType`.
2. Napišite razred `Token`.
3. Napišite kostur razreda `Lexer`. Neka metode `nextToken` i `getToken` vraćaju `null`.
4. Skinite pripremljenu datoteku `prob1tests.zip`. Sadržaj te datoteke raspakirajte u direktorij projekta u koji stavljate testove. Potom u Eclipsu napravite desni klik na projekt i "Refresh". Eclipse bi morao postati svjestan da je dobio nove izvorne kodove.

Otvorite u Eclipsu razred `Prob1Test` koji ste dobili u sklopu testova iz ZIP arhive (bit će u paketu `hr.fer.zemris.java.hw03.prob1`). Pripremio sam niz JUnit testova koje Eclipse zna pokretati. Svaka metoda koja je u tom razredu označena anotacijom `@Test` predstavlja jedan test. Inicijalno, uz sve sam metode napisao i anotaciju `@Disabled` kojom sustav pri izvođenju testova te testove preskače.

Svaki od testova predstavlja jedan jednostavan primjer uporabe razreda `Lexer` i njegovog očekivanog ponašanja. Testove sam pisao od jednostavnijih prema složenijima. Sama datoteka je podijeljena u dva dijela gdje je čitav drugi dio zakomentiran.

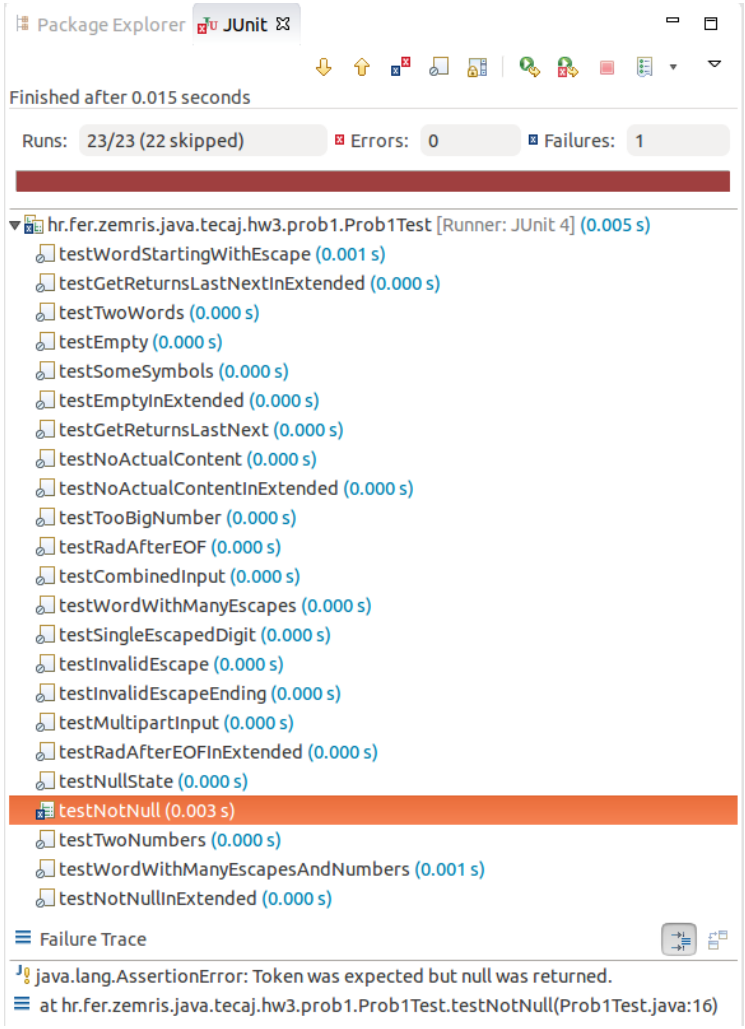
Uz datoteku s testovima bez ikakvih izmjena, pokrenite sve testove (desni klik u Package Exploreru na razred `Prob1Test` pa "Run As" → "JUnit Test". Na dijelu gdje je bio Package Explorer otvorit će se nova kartica naziva "JUnit" s rezultatima testiranja. Svi testovi bit će označeni kao preskočeni (ali neće biti

nikakvih pogrešaka, sve se može prevesti i pokrenuti).

Sada krenite redom: obrišite anotaciju `@Disabled` uz prvi test, snimite datoteku pa pokrenite sve testove.

U prikazanom izvještaju (vidi sliku u nastavku) imat ćete taj test označen kao test koji nije prošao: test stvara `Lexer` s praznim nizom i očekuje da će prvi poziv metode `nextToken` vratiti token; međutim, metoda vraća `null` i test pada. Krenite polako s izradom metode `nextToken` i riješite uočeni problem. Nastavite dalje – napišite tu metodu kako mislite da je dobro i malo po malo omogućavajte daljnje testove i pokrećite ih.

Tek kada riješite sve što je u datoteci s testovima napisano (a da nije zakomentirano – vidjet ćete taj dio, posebno je označen), nastavite dalje s ovim dokumentom.



(prethodna slika prikazuje pogrešan naziv paketa – zanemarite)

Ako ste riješili sve testove koji nisu bili zakomentirani (prvi dio datoteke s testovima), sada ste spremni za dalje. Malo ćemo promijeniti pravila igre za lexer.

Ulazna datoteka može se sastojati od različitih dijelova teksta koji se obrađuju u tokene po različitim pravilima. Da ne kompliciramo, radit ćemo s primjerom gdje postoje dvije vrste pravila.

- Tekst se sve do pojave simbola `'#'` obrađuje kako je prethodno pojašnjeno.
- Pojavom simbola `'#'` prelazi se u režim rada u kojem lexer sve uzastopne nizove bilo slova bilo

znakova bilo simbola tretira kao jednu riječ; također, ne postoji escapanje odnosno pojava znaka `\` ne predstavlja ništa drugačije u odnosu na pojavu bilo kojeg drugog znaka. U tom dijelu riječi su međusobno razdijeljene prazninama. U ovom režimu lexer nikada ne generira token tipa `NUMBER`. Ovo se područje proteže sve do pojave sljedećeg znaka `'#'` (koje generira token tipa simbol). Izvan tog područja ponovno sve ide po starom – do pojave novog ovakvog područja.

Da bismo omogućili pozivatelju za kontrolira u kojem je stanju lexer odnosno po kojim pravilima radi, definirajte novu enumeraciju `LexerState` koja ima konstante `BASIC` (predstavlja osnovni način obrade) te `EXTENDED` (predstavlja prošireni način obrade – naše drugo područje).

Potom proširite razred `Lexer` tako da mu dodate metodu:
`public void setState(LexerState state) {…}`
kojom pozivatelj izvana može postaviti željeni način grupiranja (te ako još što trebate zbog toga – učinite to). Inicijalno, lexer se mora konstruirati u stanju `BASIC`.

Kako sada vanjski klijent koristi `Lexer`? Ideja je jednostavna: sve dok lexer ne vrati token tipa simbol sadržaja '#', pozivatelj konzumira tokene u skladu s njihovom semantikom. Kad dobije token tipa simbol sadržaja '#', pozivatelj lexer prebaci u drugo stanje pozivom upravo dodane metode `setState`. Nastavlja konzumirati tokene koje proizvodi lexer ali oni se sada stvaraju po drugim pravilima. Kad ponovno dobije token tipa simbol sadržaja '#', pozivatelj lexer vrati u prvo stanje pozivom metode `setState` i nastavi dalje konzumirati tokene. Ovaj mehanizam trebat ćete i u sljedećem zadatku u kojem ćete u različitim područjima potencijalno generirati i različite skupove tokena te raditi escapeanje po različitim pravilima (konkretno: jedna pravila će vrijediti u tekstu, a druga u tagu).

Kad ste to napravili, sada odkomentirajte drugi dio testova, omogućavajte ih jednog po jednog i prepravite implementaciju tako da sve proradi odnosno da svi testovi prođu.

Problem 3.

Now you are ready for something more useful which you will actually use in one of the following homeworks in second part of semester.

We will write two hierarchies of classes: *nodes* and *elements*. Place the classes into packages `hr.fer.zemris.java.custom.scripting.elems` and `hr.fer.zemris.java.custom.scripting.nodes` respectively. *Nodes* will be used for representation of structured documents. *Elements* will be used to for the representation of expressions. If you wish, you can add support for the *Visitor Design Pattern* (see book) but other then that, `Element` and `Node` classes should offer just public methods defined here.

`Element` hierarchy

`Element` – base class having only a single public function: `String asText();` which for this class returns an empty String.

`ElementVariable` – inherits `Element`, and has a single read-only[1] `String` property: `name`. Override `asText()` to return the value of `name` property.

`ElementConstantInteger` – inherits `Element` and has single read-only `int` property: `value`. Override `asText()` to return string representation of `value` property.

`ElementConstantDouble` – inherits `Element` and has single read-only `double` property: `value`. Override `asText()` to return string representation of `value` property.

`ElementString` – inherits `Element` and has single read-only `String` property: `value`. Override `asText()` to return `value` property.

`ElementFunction` – inherits `Element` and has single read-only `String` property: `name`. Override `asText()` to return `name` property.

`ElementOperator` – inherits `Element` and has single read-only `String` property: `symbol`. Override `asText()` to return `symbol` property.

`Node` hierarchy

`Node` – base class for all graph nodes.

`TextNode` – a node representing a piece of textual data. It inherits from `Node` class.

`DocumentNode` – a node representing an entire document. It inherits from `Node` class.

`ForLoopNode` – a node representing a single for-loop construct. It inherits from `Node` class.

`EchoNode` – a node representing a command which generates some textual output dynamically. It inherits from `Node` class.

Lets assume that we work with following text document:

1 If class has property Prop, this means that it has private instance variable of the same name and the public getter method (getProp()) and the public setter method (setProp(value)). If property is read-only, no setter is provided. If property is write-only, no getter is provided. For read-only properties, use constructor to initialize it.

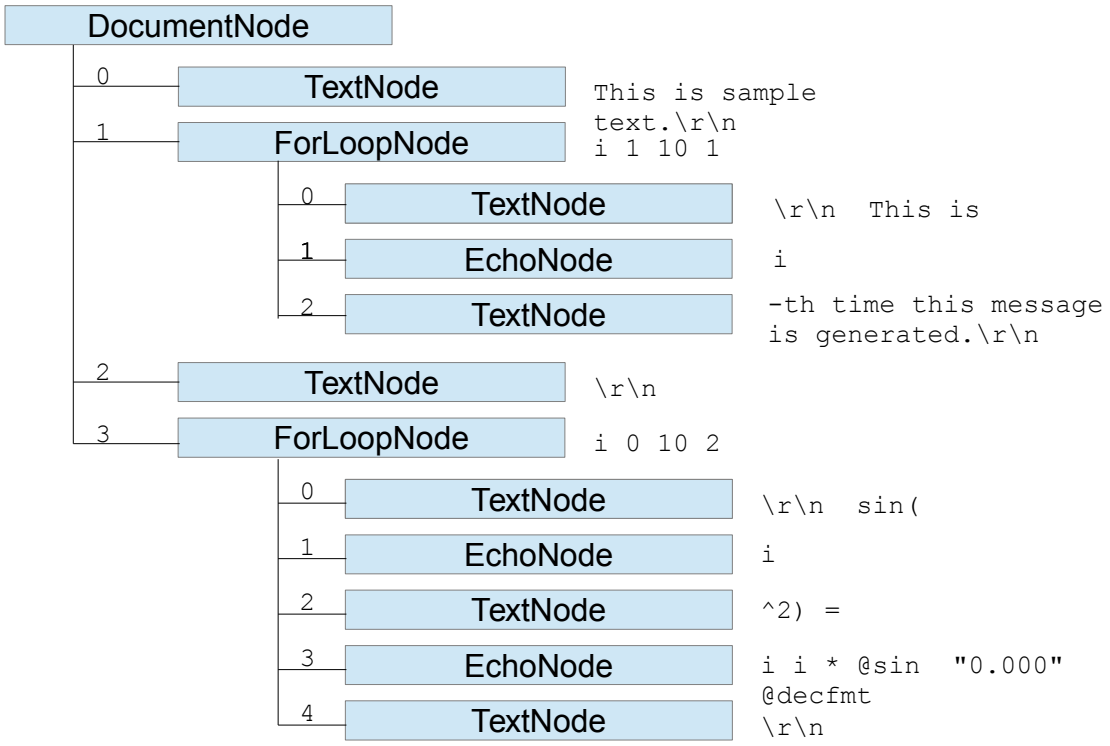
```
This is sample text.
{$ FOR i 1 10 1 $}
  This is {$= i $}-th time this message is generated.
{$END$}
{$FOR i 0 10 2 $}
  sin({$=i$}^2) = {$= i i * @sin  "0.000" @decfmt $}
{$END$}
```

This document consists of tags (bounded by `{$` and `$}`) and rest of the text. Reading from top to bottom we have:

text	`This is sample text.\r\n`	1
tag	`{$ FOR i 1 10 1 $}`	2
text	`\r\n  This is`	3
tag	`{$= i $}`	4
text	`-th time this message is generated.\r\n`	5
tag	`{$END$}`	6
text	`\r\n`	7
tag	`{$FOR i 0 10 2 $}`	8
text	`\r\n  sin(`	9
tag	`{$=i$}`	10
text	`^2) =`	11
tag	`{$= i i * @sin  "0.000" @decfmt $}`	12
text	`\r\n`	13
tag	`{$END$}`	14

Observe that spaces in tags are ignorable; `{$END$}` means the same as `{$ END  $}`. Each tag has its name. The name of `{$ FOR … $}` tag is `FOR`, and the name of `{$= … $}` tag is `=`. Tag names are case-insensitive. This means that you can write `{$ FOR … $}` or `{$ For … $}` or `{$ foR … $}` or similar. A one or more spaces (tabs, enters or spaces – we will treat them equally) can be included before tag name, so all of the following is also OK: `{$FOR … $}` or `{$ FOR … $}` or `{$  FOR … $}`. `=`-tag is an empty tag – it has no content so it does not need closing tag. `FOR`-tag, however, is not an empty tag. Its has content and an accompanying `END`-tag must be present to close it. For example, the content of the `FOR`-tag opened in the line 2 in above table comprises two texts and a tag given in lines 3, 4 and 5. Since `END`-tag is only here to help us close nonempty tags, it will not have its own representation.

The document model built from this document looks as follows.



Class `Node` defines methods:

`void addChildNode(Node child);` – adds given `child` to an internally managed collection of children; use an instance of `ArrayIndexedCollection` class for this. However, create this collection only when actually needed (i.e. create an instance of the collection on demand → on first call of `addChildNode`).

`int numberOfChildren();` – returns a number of (direct) children. For example, in above example, instance of `DocumentNode` would return 4.

`Node getChild(int index);` – returns selected child or throws an appropriate exception if the index is invalid.

All other node-classes inherit from `Node` class.
Class `TextNode` defines single additional read-only String property `text`.

Class `ForLoopNode` defines several additional read-only properties:

- property `variable` (of type `ElementVariable`)
- property `startExpression` (of type `Element`)
- property `endExpression` (of type `Element`)
- property `stepExpression` (of type `Element`, which can be `null`)

Class `EchoNode` defines a single additional read-only `Element[]` property `elements`.

As you can see, `ForLoopNode` and `EchoNode` work with instances of `Element` (sub)class. Lets take a look on =-tag from our example:

```
{$= i i * @sin  "0.000" @decfmt $}
```

Arguments (parameters) of this tag are:

- two times `ElementVariable` with `name="i"`
- once `ElementOperator` with `symbol="*"`
- once `ElementFunction` with `name="sin"`
- once `ElementString` with `value="0.000"`
- once `ElementFunction` with `name="decfmt"`

Implement a parser for described structured document format. Implement it as a class `SmartScriptParser` and put it in the package `hr.fer.zemris.java.custom.scripting.parser`. For this parser implement appropriate lexer and put it in the package `hr.fer.zemris.java.custom.scripting.lexer`.
The parser should have a single constructor which accepts a string that contains document body. In this constructor, parser should create an instance of lexer and initialize it with obtained text. The parser should use lexer for production of tokens. The constructor should delegate actual parsing to separate method (in the same class). This will allow us to later add different constructors that will retrieve documents by various means and delegate the parsing to the same method. Create a class `SmartScriptParserException` (derive it from `RuntimeException`) and place it in the same package as `SmartScriptParser`. If any exception occurs during parsing, parser should catch it and rethrow an instance of this exception.

Important
Problem 1 in this homework was an illustration of lexer creation. Here you are creating a separate lexer with different token types, different set of rules for character grouping (see below for some specification) etc. You are expected to place all relevant classes and enums in new package (as defined above). Your parser must use this lexer and not perform character grouping by itself. Take some time to think about:

- how many different token types we need here, in order to be able to create simple parser;
- how many states do we need in lexer?

Please observe that tag `ForLoopNode` can have three or four parameters (as specified by user): first it must have one `ElementVariable` and after that two or three `Element`s of type variable, number or string. If user specifies something which does not obeys this rule, throw an exception. Here are several good examples:

```
{$ FOR i -1 10 1 $}
{$    FOR    sco_re             "-1"10 "1" $}
{$ FOR year 1 last_year $}
```

Please observe that `"-1"10` is OK: is should parse into string and after it into integer. Here, lexer knows how string ends, so after it, first character can start new token.
Here are several bad examples (for which an exception should be thrown; explanation is shown on right side and is not part of example):

```
{$ FOR 3 1 10 1 $}                  // 3 is not variable name
{$ FOR * "1" -10 "1" $}             // * is not variable name
{$ FOR year @sin 10 $}              // @sin function element
{$ FOR year 1 10 "1" "10" $}        // too many arguments
{$ FOR year $}                      // too few arguments
{$ FOR year 1 10 1 3 $}             // too many arguments
```

Valid variable name starts by letter and after follows zero or more letters, digits or underscores. If name is not valid, it is invalid. This variable names are valid: `A7_bb`, `counter`, `tmp_34`; these are not: `_a21`, `32`, `3s_ee` etc.

Valid function name starts with `@` after which follows a letter and after than can follow zero or more letters, digits or underscores. If function name is not valid, it is invalid.

Valid operators are + (plus), - (minus), * (multiplication), / (division), ^ (power).

Valid tag names are "=", or variable name. So = is valid tag name (but not valid variable name).

When parsing content inside tag, lexer should extract as many characters as possible into each token. Spaces between tokens are ignorable and generally not required. Decimal numbers are only recognized in digits-dot-digits format, but not in scientific notation. The consequence of this is that tag:

```
{$ FOR i-1.35bbb"1" $}
```

is correct, and is tokenized the same way as if the following was given:

```
{$ FOR i -1.35 bbb "1" $}
```

In lexer, when deciding what to do with minus sign, treat it as a symbol if immediately after it there is no digit. Only when immediately after it (no spaces between) a digit follows (lexer can check this!), treat it as part of negative number.

In strings which are part of tags (*and only in strings*!) parser must accept the following escaping:
`\\` sequence treat as a single string character `\`
`\"` treat as a single string character `"` (and not the end of the string)
`\n`, `\r` and `\t` have its usual meaning (ascii 10, 13 and 9).
Every other sequence which starts with `\` should be treated as invalid and throw an exception.
For example, `"Some \\ test X"` should be interpreted as string with value `Some \ test X`. Another example: `"Joe \"Long\" Smith"` represents a single string with value `Joe "Long" Smith`. Please note: string `"Some \\ test X"` shown here is what user would write in his document, which will be analyzed. If you wish to write it in Java source code (e.g. for testing purposes), you will have to escape each " and each \. Here is example:

String simpleDoc = `"A tag follows {$= \"Joe \\\"Long\\\" Smith\"$}."`

which will represent document:

```
A tag follows {$= "Joe \"Long\" Smith"$}.
```

In document text (i.e. **outside of tags**) parser must accept only the following two escaping:
`\\` treat as `\`
`\{` treat as `{`
Every other sequence which starts with `\` should throw an exception. Please note, in this context, character { is just a regular character if it is not followed directly by $. So the next document is just a single text:

```
Example { bla } blu \{$=1$}. Nothing interesting {=here}.
```

For example, document whose content is following:

```
Example \{$=1$}. Now actually write one {$=1$}
```

should be parsed into only three nodes:

```
DocumentNode
  *
  *- TextNode with value Example {$=1$}. Now actually write one
  *- EchoNode with one element
```

Implementation hint. As help for tree construction use `ObjectStack` from your previous homework – copy in this homework all needed files (but nothing more) so that this code could compile on reviewers computers later. At the beginning, push `DocumentNode` to stack. Then, for each empty tag or text node create that tag/node and add it as a child of `Node` that was last pushed on the stack. If you encounter a non-empty tag (i.e. `FOR`-tag), create it, add it as a child of `Node` that was last pushed on the stack and than push this `FOR`-node to the stack. Now all nodes following will be added as children of this `FOR`-node; the exception is `{$END$}`; when you encounter it, simple pop one entry from the stack. If stack remains empty, there is error in document – it contains more `{$END$}`-s than opened non-empty tags, so throw an exception.

During the tag construction, you do not have to consider whether the provided tags are meaningful. For example, in tag:

```
{$= i i * @sin  "0.000" @decfmt $}
```

you do not have to think about is it OK that after two variables `i` comes the `*`-operator. You task for now is just to build the accurate document model which represents the document **as provided by the user**. At some later time we will consider whether that which user gave us is actually legal or not.

The developed parser should be used as illustrated by the following scriptlet:

```
String docBody = "....";
SmartScriptParser parser = null;
try {
  parser = new SmartScriptParser(docBody);
} catch(SmartScriptParserException e) {
  System.out.println("Unable to parse document!");
  System.exit(-1);
} catch(Exception e) {
  System.out.println("If this line ever executes, you have failed this class!");
  System.exit(-1);
}
DocumentNode document = parser.getDocumentNode();
String originalDocumentBody = createOriginalDocumentBody(document);
System.out.println(originalDocumentBody); // should write something like original
                                          // content of docBody
```

Create a main program named `SmartScriptTester` and place it in package `hr.fer.zemris.java.hw03`. In the `main` method put the above-shown scriptlet. Let this program accepts a single command-line argument: path to document. You can read the content of this file by following code:

```
import java.nio.file.Files;
import java.nio.charset.StandardCharsets;
import java.nio.file.Paths;
String docBody = new String(
                  Files.readAllBytes(Paths.get(filepath)),
                  StandardCharsets.UTF_8
);
```

In your project create directory examples and place inside at least `doc1.txt` which contains the example given in this document. You are free to add more examples.

Implement all needed methods in order to ensure that the program works.

The method `createOriginalDocumentBody` does not have to reproduce the exact original documents, since this is impossible: after the parsing is done you have lost the information how some of the elements were separated (by one or more spaces, tabs, etc. and similar). But it must reproduce something which will after parsing again result with the same document model! So this is the actual test:

```
String docBody = "....";
SmartScriptParser parser = new SmartScriptParser(docBody);
DocumentNode document = parser.getDocumentNode();
String originalDocumentBody = createOriginalDocumentBody(document);
SmartScriptParser parser2 = new SmartScriptParser(originalDocumentBody);
DocumentNode document2 = parser2.getDocumentNode();
// now document and document2 should be structurally identical trees
```

Please note: for testing purposes, you can prepare a lot of simple documents and save them into src/test/resources (create this directory, Refresh project and also right click on directory, Maven → Update Project; after that you will see src/test/resources shown as source-folder). Lets say you have file document1.txt directly in src/test/resources (you can create it in Eclipse using New→File wizard), so it's actual path is src/test/resources/document1.txt. You can load it and get as string using following code:

```
private String loader(String filename) {
	ByteArrayOutputStream bos = new ByteArrayOutputStream();
	try(InputStream is =
		 this.getClass().getClassLoader().getResourceAsStream(filename)) {
		byte[] buffer = new byte[1024];
		while(true) {
			int read = is.read(buffer);
			if(read<1) break;
			bos.write(buffer, 0, read);
		}
		return new String(bos.toByteArray(), StandardCharsets.UTF_8);
	} catch(IOException ex) {
		return null;
	}
}
```

The previous code is independent on actual test resources path (is it src/test/resources or something else); if we fix file path, a more simpler code is possible but this one will suffice for now.

Using this private method (e.g. in your test), you can read file with code such as:

```
String document = loader("document1.txt");
```

Be careful; if you add these files or edit them outside of Eclipse, you will have to Refresh Eclipse project in order for Eclipse to recognize them.

Very important: you *do not have to* develop an engine that will "execute" this document (iterate for-loop for specified number of iterations etc). All you have to do at this point is write a piece of code that will produce a document tree model.

You are expected to write at lease some basic junit tests for this lexer and parser.

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); the exception is class `ObjectStack` and dependent classes as specified in the *implementation hint* section of Problem 3, as well as the classes specified in Problem 1. You can not use any of Java Collection Framework classes which represent collections or its derivatives (its OK to use Arrays class if you find it suitable). Document your code!

The consultations are at standard times. Feel free to drop by my office after email announcement.

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

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. Deadline is March 28th 2019. at 07:00 AM.