Rešitev desete domače naloge (Urejanje preglednice)

Nalogo je mogoče rešiti na zelo različne načine, mi pa se je bomo lotili tako, da bomo na podlagi kriterijev urejanja zgradili primerjalnik za medsebojno primerjavo vrstic preglednice, nato pa bomo preglednico uredili s pomočjo izdelanega primerjalnika.

Hierarhija razredov za predstavitev elementov preglednice

Preglednica bi lahko bila dvodimenzionalna tabela, ker pa se generiki s tabelami ne ujemajo najbolje, bomo raje izdelali seznam vrstic, vsaka vrstica pa bo seznam elementov. Elemente bi lahko hranili kot objekte tipa `Integer` in `String` (v tem primeru bi bila preglednica tipa `List<List<Object>>`, ker je `Object`, kot vemo, skupni nadtip tipov `Integer` in `String`), vendar pa nam bo lastna hierarhija olajšala gradnjo primerjalnikov. Preglednico bomo tako ustvarili kot objekt tipa `List<List<Element>>`, pri čemer za vse objekte tipa `Element` velja, da jih je med seboj mogoče naravno primerjati:

```
public abstract class Element implements Comparable<Element> {
}
```

Razred `Element` ne vsebuje ničesar drugega. Ker ne implementira abstraktne metode `compareTo`, mora biti tudi sam abstrakten. Njegovi podrazredi ne bodo abstraktni, zato bodo morali implementirali metodo

`public int compareTo(Element drugi)`.

Razred `Element` bo imel tri podrazrede: `ElementString` (za besedilne elemente preglednice), `ElementInt` (za celoštevilske elemente) in `ElementDaNeMorda` (za elemente vrste da-ne-morda). Razreda `ElementString` in `ElementInt` sta preprosta ovojnika za spremenljivko tipa `String` oziroma `int`, omeniti velja le implementacijo metode `compareTo`.

```
public class ElementString extends Element {
    private String vrednost;

    public ElementString(String vrednost) {
        this.vrednost = vrednost;
    }

    @Override
    public int compareTo(Element drugi) {
        ElementString drugiE = (ElementString) drugi;
        return this.vrednost.compareTo(drugiE.vrednost);
    }

    @Override
    public String toString() {
        return this.vrednost;
    }
}
```

```
public class ElementInt extends Element {
    private int vrednost;

    public ElementInt(int vrednost) {
        this.vrednost = vrednost;
    }

    @Override
    public int compareTo(Element drugi) {
        ElementInt drugiE = (ElementInt) drugi;
        return this.vrednost - drugiE.vrednost;
    }

    @Override
    public String toString() {
        return Integer.toString(this.vrednost);
    }
}
```

V metodi `compareTo` bi morali pred uporabo pretvorbe tipa preveriti, ali je objekt `drugi` ciljnega tipa. Vendar pa ni povsem jasno, kaj naj bi metoda vrnila, če se tipa ne bi ujemala. Najbolje bi bilo sprožiti izjemo, to pa se bo v takem primeru tako ali tako zgodilo.

Razred `ElementDaNeMorda` je za odtenek zanimivejši. Ker je neposredna primerjava nizov `da`, `ne` in `morda` okorna, jih pretvorimo v indekse 0, 1 in 2, te pa lahko primerjamo na enak način kot števila.

```
import java.util.List;

public class ElementDaNeMorda extends Element {
    private static final List<String> NIZI = List.of("da", "ne", "morda");
    private int indeks;

    public ElementDaNeMorda(String vrednost) {
        this.indeks = NIZI.indexOf(vrednost);
    }

    @Override
    public int compareTo(Element drugi) {
        ElementDaNeMorda drugiE = (ElementDaNeMorda) drugi;
        return this.indeks - drugiE.indeks;
    }

    @Override
    public String toString() {
        return NIZI.get(this.indeks);
    }
}
```

Namesto seznama nizov bi bilo še bolje uporabiti slovar, ki niz neposredno preslika v indeks.

Razred s statičnimi metodami za gradnjo primerjalnikov

Pri gradnji primerjalnika za urejanje preglednice bomo uporabljali različne operacije. Primerjalnike bomo po potrebi obračali, jih sestavljali ipd. Vse metode bodo parametrizirane z generičnim tipom in dovolj splošne, da jih bomo lahko uporabljali tudi za katere druge namene.

Osnovni gradniki vseh izdelanih primerjalnikov so primerjalniki, ki elemente posameznih tipov primerjajo po njihovem naravnem vrstnem redu. Takšne primerjalnike bomo pridobili z metodo

`public static <T extends Comparable<T>> Comparator<T> naravni()`,

ki za podani tip `T`, ki mora biti podtip tipa `Comparable<T>` (tej zahtevi ustreza razred `Element` in vsi njegovi podrazredi), vrne primerjalnik, čigar metoda `compare` primerja podana objekta tipa `T` na enak način kot metoda `compareTo`, ki jo implementira tip `T`.

```
import java.util.*;

public class Primerjalniki {

    public static <T extends Comparable<T>> Comparator<T> naravni() {
        return new Naravni<T>();
    }

    private static class Naravni<T extends Comparable<T>>
                         implements Comparator<T> {
        @Override
        public int compare(T a, T b) {
            return a.compareTo(b);
        }
    }
    ...
}
```

Našo preglednico, objekt tipa `List<List<Element>>`, bomo urejali po vrsticah (objektih tipa `List<Element>`), ne po posameznih elementih. To pomeni, da bomo delali s primerjalniki, ki med seboj primerjajo istoležna elementa v objektih tipa `List<Element>`. Sledeča metoda vrne primerjalnik, ki s primerjalnikom `prim` primerja podana objekta tipa `List<T>` po njunih elementih na indeksu `indeks`:

```
public class Primerjalniki {
    ...
    public static <T> Comparator<List<T>> poElementihNaIndeksu(
            Comparator<T> prim, int indeks) {

        return new PoElementihNaIndeksu<T>(prim, indeks);
    }

    private static class PoElementihNaIndeksu<T>
                    implements Comparator<List<T>> {
        Comparator<T> prim;
        int indeks;
```

```
        public PoElementihNaIndeksu(Comparator<T> prim, int indeks) {
            this.prim = prim;
            this.indeks = indeks;
        }

        @Override
        public int compare(List<T> a, List<T> b) {
            return this.prim.compare(a.get(this.indeks), b.get(this.indeks));
        }
    }
    ...
}
```

Kadar bodo navodila zahtevala urejanje v obratnem vrstnem redu, si bomo pomagali s primerjalnikom, ki ga bomo zgradili kot obrat že izdelanega primerjalnika. Metoda `obrat` na podlagi primerjalnika `prim` izdela primerjalnik, ki podana objekta `a` in `b` primerja tako, kot primerjalnik `prim` primerja objekta `b` in `a`.

```
public class Primerjalniki {
    ...
    public static <T> Comparator<T> obrat(Comparator<T> prim) {
        return new Obrat<T>(prim);
    }

    private static class Obrat<T> implements Comparator<T> {
        Comparator<T> prim;

        public Obrat(Comparator<T> prim) {
            this.prim = prim;
        }

        @Override
        public int compare(T a, T b) {
            return this.prim.compare(b, a);
        }
    }
    ...
}
```

Potrebovali bomo tudi metodo za sestavljanje primerjalnikov. Metoda `kompozitum` na podlagi primerjalnikov `prim1` in `prim2` izdela primerjalnik, ki podana objekta primerja s primerjalnikom `prim1`, če se ne razlikujeta, pa še po primerjalniku `prim2`.

```
public class Primerjalniki {
    ...
    public static <T> Comparator<T> kompozitum(
                          Comparator<T> prim1, Comparator<T> prim2) {
        return new Kompozitum<T>(prim1, prim2);
    }

    private static class Kompozitum<T> implements Comparator<T> {
```

```
        Comparator<T> prim1;
        Comparator<T> prim2;

        public Kompozitum(Comparator<T> prim1, Comparator<T> prim2) {
            this.prim1 = prim1;
            this.prim2 = prim2;
        }

        @Override
        public int compare(T a, T b) {
            int primerjava = this.prim1.compare(a, b);
            return (primerjava == 0) ? (this.prim2.compare(a, b)) :
                                       (primerjava);
        }
    }
    ...
}
```

Glavni razred

V glavnem razredu najprej preberemo podatke o stolpcih preglednice, urejevalne kriterije in sámo preglednico.

```
import java.util.*;

public class Urejanje {

    private static final int TIP_INT = 1;
    private static final int TIP_STRING = 2;
    private static final int TIP_DA_NE_MORDA = 3;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int stVrstic = sc.nextInt();
        int stStolpcev = sc.nextInt();

        // stolpci
        int[] tipi = new int[stStolpcev];
        for (int i = 0;  i < stStolpcev;  i++) {
            tipi[i] = sc.nextInt();
        }

        // kriteriji
        int stKriterijev = sc.nextInt();
        int[] kriteriji = new int[stKriterijev];
        for (int i = 0;  i < stKriterijev;  i++) {
            kriteriji[i] = sc.nextInt();
        }

        // preglednica
        List<List<Element>> preglednica = new ArrayList<>();
```

```
        for (int i = 0;  i < stVrstic;  i++) {
            List<Element> vrstica = new ArrayList<>();
            for (int j = 0;  j < stStolpcev;  j++) {
                switch (tipi[j]) {
                    case TIP_INT:
                        vrstica.add(new ElementInt(sc.nextInt()));
                        break;

                    case TIP_STRING:
                        vrstica.add(new ElementString(sc.next()));
                        break;

                    case TIP_DA_NE_MORDA:
                        vrstica.add(new ElementDaNeMorda(sc.next()));
                        break;
                }
            }
            preglednica.add(vrstica);
        }
        ...
    }
    ...
}
```

Zatem na podlagi kriterijev zgradimo primerjalnik in ga posredujemo metodi za urejanje.

```
public static void main(String[] args) {
    ...
    preglednica.sort(izdelajPrimerjalnik(kriteriji));
    ...
}
```

Nazadnje le še izpišemo urejeno preglednico.

```
public static void main(String[] args) {
    ...
    for (List<Element> vrstica: preglednica) {
        boolean prvic = true;
        for (Element element: vrstica) {
            if (prvic) {
                prvic = false;
            } else {
                System.out.print("|");
            }
            System.out.print(element);
        }
        System.out.println();
    }
}
```

Ostane nam še metoda `izdelajPrimerjalnik`. Za vsak kriterij oziroma pripadajoči stolpec preglednice izdelamo primerjalnik `prim0`, ki podana objekta (ta pripadata istemu podtipu

tipa `Element`) primerja po njuni naravni urejenosti. Na podlagi primerjalnika `prim0` nato izdelamo primerjalnik `prim`, ki med seboj primerja seznama tipa `List<Element>`, in sicer po njunih elementih na podanem indeksu. Primerjalnik `prim` po potrebi še obrnemo. Dokončni primerjalnik zgradimo kot kompozitum vseh izdelanih primerjalnikov `prim`.

```
public class Urejanje {
    ...
    private static Comparator<List<Element>> izdelajPrimerjalnik(int[] kriteriji) {
        Comparator<List<Element>> primerjalnik = null;

        for (int i = 0;  i < kriteriji.length;  i++) {
            Comparator<Element> prim0 = Primerjalniki.naravni();

            int stolpec = Math.abs(kriteriji[i]) - 1;
            Comparator<List<Element>> prim =
                Primerjalniki.poElementihNaIndeksu(prim0, stolpec);

            if (kriteriji[i] < 0) {
                prim = Primerjalniki.obrat(prim);
            }

            if (primerjalnik == null) {
                primerjalnik = prim;
            } else {
                primerjalnik = Primerjalniki.kompozitum(primerjalnik, prim);
            }
        }
        return primerjalnik;
    }
}
```

Nekoliko več pozornosti si zasluži sledeči stavek:

```
Comparator<Element> prim0 = Primerjalniki.naravni();
```

Kako prevajalnik ugotovi, s kakšnim tipom mora zamenjati generični tip `T`? Ker se rezultat klica metode priredi spremenljivki tipa `Comparator<Element>`, je jasno, da se mora tip `T` nadomestiti s tipom `Element`. Nadomestni tip bi lahko ob klicu metode `naravni` tudi eksplicitno podali ...

```
Comparator<Element> prim0 = Primerjalniki.<Element>naravni();
```

... vendar pa lahko prevajalnik nadomestne tipe pri generičnih metodah v veliki večini primerov izlušči sam.