Laborator 5

Exercitiu

Manipularea sirurilor de caractere si expresii regulate

Tema

Exercitiu

Definiti o structura de clase care sa reflecte urmatoarea situatie:
O institutie are mai multi angajati de diferite tipuri.

1. Profesor, care are un nume, prenume, vechime, disciplina, grad academic (un numar intreg, 1,2,3,4,5), salariu de baza. Unui profesor i se poate calcula salariul (functia getSalariu) dupa formula: Salariu de baza + SPOR_PROFESOR*vechime + SPOR_GRAD*grad, unde SPOR_PROFESOR si SPOR_GRAD sunt niste constante specifice profesorilor.
2. Medic, care are un nume, prenume, vechime, specialitate, garzi efectuate si un salariu de baza. Unui medic i se poate calcula salariul (functia getSalariu) dupa formula: Salariu de baza*factorGarda() + SPOR_GARDA*garzi, unde SPOR_GARDA este o constanta iar factorGarda este o alta functie care returneaza un numar in virgula mobila dupa formula daca specialitate == “Ortopedie” -> 1+vechime/10. Daca specialitate == “Neurologie” -> 1 + vechime/8. Daca specialitate == “Nefrologie” -> 1+vechime/12. Altfel -> 1 + vechime/11.
3. Inginer, care are nume, prenume, vechime si un salariu de baza. Unui inginer i se poate calcula salariul (functia getSalariu) folosind formula salariuDeBaza * (1+vechime/16).
4. Administrator, care are nume, prenume si care are un salariu fix (constant in cod, tot functia getSalariu).

Optional: integrati cu frameworkul din laboratorul 4.

Declarati un vector cu tipul cel mai generic. Care e acesta?. Folositi o bucla for pentru a afisa o lista cu “Nume, Prenume -> Salariu”.

Manipularea sirurilor de caractere si expresii regulate

Sirurile de caractere sunt tratate in java ca un obiect special de tip String. Pentru usurinta utilizarii sirurilor de caractere exista operatorul “+” care permite concatenarea cu usurinta a sirurilor de caractere. Desi de cele mai multe ori concatenarea in acest fel a sirurilor de caractere este acceptabila, pentru programele care folosesc aceasta operatie foarte frecvent e

posibil ca operatia in sine sa devina extrem de costisitoare. Pentru a intelege acest lucru, trebuie sa vedem ce se intampla in realitate atunci cand concatenam doua stringuri folosind operatorul “+”.

```
String a = “a”;
String b = “b”;
String result = a+b;
```

Mai intai, “a” este un sir de caractere constant la compilare. Din cauza ca este constant la compilare, “a” va fi stocat in memorie intr-o zona speciala numit String Pool. Similar cu b.
In momentul in care facem a+b, de fapt este construit un nou obiect de tip String care la constructie primeste valoarea “ab”.
Pana acum nimic rau, totul pare ok. Sa luam acum alt caz:

```
String a = “a”;
String b = “b”;
String c = “c”;
String d = “d”;
String e = “e”;
String f = “f”;
String result = a+b+c+d+e+f;
```

De aceasta data, lucrurile nu mai sunt la fel de bune. Calculatorul nu stie sa evalueze a+b+c+d+e+f ca o singura operatie ci ca o lista de operatii individuale si de aici apar problemele. Expresia de mai sus practic este evaluata asa:

```
result = a+b+c+d+e+f
result = ((((a+b)+c)+d)+e)+f;
result = (((“ab”+c)+d)+e)+f;
result = ((“abc”+d)+e)+f;
result = (“abcd”+e)+f;
result = “abcde”+f;
result = “abcdef”;
```

Observam ca de data aceasta sunt construite o multime de obiecte intermediare “ab”, “abc”, “abcd” etc. Aceasta constructie implica niste overhead, ocupa memorie, care apoi e eliberata de garbage collector. Acele obiecte intermediare nu ne intereseaza insa. Pentru aceste cazuri, au fost adaugate doua clase care ne ajuta sa manipulam obiecte String fara sa construim noi obiecte intermediare. Operatia de mai sus poate fi scrisa similar asa:

```
String a = “a”;
String b = “b”;
String c = “c”;
String d = “d”;
String e = “e”;
String f = “f”;
StringBuffer buf = new StringBuffer();
String result = buf.append(a).
                    append(b).
                    append(c).
```

```
                append(d).
                append(e).
                append(f).toString();
```
In aceasta situatie, obiectele intermediare nu mai sunt generate.
Evident, diferenta de performanta este foarte mica si probabil veti simti aceasta diferenta daca aplicatia lucreaza efectiv cu siruri de caractere care trebuie concatenate frecvent. Un exemplu ar fi daca vrem sa “spargem” un sir de caractere in bucati mai mici, sa eliminam ce nu ne place sau sa schimbam si apoi sa recompunem sirul mare, totul intr-o bucla, am vrea ca raspunsul sa fie aproape instant si ne asteptam sa primim multe cereri de acest fel pe secunda (ca un server web?). Atunci, operatia de concatenare simpla ar fi o alegere foarte proasta.
Exista doua clase care fac acest lucru, una este StringBuffer si una este StringBuilder. Cele doua sunt identice, dar metodele din StringBuffer sunt sincronizate (vom reveni asupra acestui aspect cand vorbim de Thread). Pe scurt, StringBuffer e thread safe, StringBuilder nu.

Pentru a concatena doua siruri putem folosi clasa StringBuffer. Pentru a sparge un sir de caractere avem insa o multitudine de metode in functie de ceea ce vrem sa facem. Sa spunem ca avem un sir de caractere care contine valori separate prin virgula.
```
String csv = “a,bg,234,asdg”;
String[] exploded = csv.split(“,”);
```
Vectorul exploded va contine 4 siruri de caractere, valorile a,bg,234,asdg.
Ceea ce este mai frumos este faptul ca putem folosi ca parametru al lui split o expresie regulata.

Descarcati fisierul lab5gui.jar de la adresa:

https://drive.google.com/drive/folders/1z0R1dEiNFP7sudM_SWrX77NPmLklBcD7

Pentru a utiliza aplicatia mai usor, am expus doua clase, ambele singleton (cum le folosim?). Clasa Console are metoda “append” care adauga un mesaj in fereastra de consola si “clear” care sterge textul din acea fereastra.
Clasa InputWindow care are metodele:

1. getText, care returneaza ce se afla in zona de text
2. getRegexp, care returneaza ce se afla in zona de regular expression
3. setSplitHandler care primeste ca parametru o instanta a clasei “Splitter”
4. setMatcherHandler care primeste ca parametru o instanta a clasei “MyMatcher”

Clasele Splitter si MyMatcher trebuie extinse de o clasa scrisa de voi si contin metode care vor fi apelate atunci cand userul apasa pe butoanele corespunzatoare din interfata.

Haideti sa facem experimente cu expresii regulate.

1. Impartiti sirul: “`Alina, Mihai     , George  ,Dan   `” eliminand in acelasi timp spatiile din fata si de dupa nume.
2. Impartiti sirul: “`Alina, Mihai: George; Dan, Diana,Maria;Octavian`” folosind ca separator oricare din caracterele ,:;

In spate, este utilizata perechea de clase Pattern si Matcher.
Pattern este folosit pentru a compila un sir de caractere intr-un tipar ce poate fi folosit de sistem.
Matcher este o clasa care ne ajuta sa jonglam cu aceste tipare.

Exemplu: vrem sa verificam daca un sir de caractere este de forma litera mare apoi mai multe litere mici, apoi o virgula, apoi un spatiu, apoi litera mare si mai multe litere mici.

```
Pattern pattern = Pattern.compile("[A-Z][a-z]+, [A-Z][a-z]+");
Matcher matcher = pattern.matcher("Mintici, Octavian");
```

matcher.find() va returna true daca exista pe undeva o secventa de caractere care sa se potriveasca cu tiparul introdus. Daca da, matcher va fi setat pe acea secventa. Pentru a gasi secventa folosim matcher.group(). O urmatoare apelare a lui matcher.find() ne va pozitiona pe urmatoarea pozitie (sau va returna false daca nu exista).
Exercitiu:

1. Folositi matcher pentru a afisa toate numele folosind conventia: un nume este un sir de caractere mici care incepe cu o litera mare.
2. Folositi matcher pentru a verifica daca un sir de caractere contine o adresa de email
3. Folositi matcher pentru a verifica daca un sir de caractere ESTE o adresa de email
4. Folositi matcher pentru a extrage o lista de adrese de email dintr-un sir de caractere

Detalii complete legate de expresiile regulate le gasiti la:
https://docs.oracle.com/javase/tutorial/essential/regex/index.html
iar lista de caractere speciale (pentru Pattern) la:
https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

Tema

Scrieti o functie care primeste ca parametru un String si intoarce true daca sirul este un document XML valid si false altfel. Pentru simplitate, consideram ca un document XML este valid daca toate marcajele sunt deschise si inchise corect. Tot pentru simplitate vom considera ca nu exista marcaje care se inchid singure.

```
<marcaj1>
<marcaj2>
</marcaj2>
</marcaj1>
```

Corect.

```
<marcaj1>
<marcaj2>
</marcaj1>
```

Gresit, marcaj 2 nu se inchide.

```
<marcaj1>
```

```
<marcaj2>
</marcaj1>
</marcaj2>
```
Gresit, marcaj 2 e deschis in marcaj1 dar se inchide dupa ce marcaj 1 se inchide.
```
<marcajCareSeInchideSingur />
```
Pentru simplitate, nu vom avea marcaje care se inchid singure.
```
<marcaj1>
      pre text
      <marcaj2>
            Text
      </marcaj2>
      <marcaj3>
            Text2
      </marcaj3>
      text
</marcaj1>
```
Si acesta este un document valid.