Wednesday, 8 March 2023

15:27

1. Изчисляване на процент:

//\*\*\*\*\*\*\*\* percentage \*\*\*\*\*\*\*\*\*\*\*\*\*\*

// System.out.printf("%.2f%%%n", small / total \* 100);

//calculating total - percentage of it

// total = (int) Math.ceil(total - (total / 100.0 \* percentage));

//double removingPercent = total \* ( percentInput / 100.0);

2.Стринг/Номер входни операции:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\* String/number input \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// double currentPiece = 0; -> could be configured outside in case of math

//actions outside the while loop;

// String input = scanner.nextLine();

//while(!input.equals("STOP")){

// currentPiece = Double.parseDouble(input);

// .....

//input = scanner.nextLine();

// }

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*another String/number input option\*\*\*\*\*\*\*\*\*\*\*

//while ( a>b){

//String input = scanner.nextLine();

//if(input.equals("String"){

// break;

// }

//int somethingElse = Integer.parseInt(input);

//{do something}

//}

3.Оператор Continue:

//\*\*\*\*\*\*\*\*\*\*\*\*need Continue example\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//String firstNum = scanner.nextLine();

//String secondNum = scanner.nextLine();

//2345

// 6789

//for (int i = firstNum.charAt(0); i <= secondNum.charAt(0); i++) {

// if (i % 2 == 0) continue;

// for (int j = firstNum.charAt(1); j <= secondNum.charAt(1); j++) {

//if (j % 2 == 0) continue;

//for (int k = firstNum.charAt(2); k <= secondNum.charAt(2); k++) {

// if (k % 2 == 0) continue;

// for (int l = firstNum.charAt(3); l <= secondNum.charAt(3); l++) {

//if (l % 2 == 0) continue;

//

//System.out.printf("%c%c%c%c ", i, j, k, l);

// String name = scanner.nextLine();

//

// double sum = 0;

// int y = 1;

// int excluded = 0;

//while (y <= 12) {

//if (excluded > 1) {

// break;

//}

//

//double grade = Double.parseDouble(scanner.nextLine());

//

//if (grade < 4) {

// excluded++;

// continue;

//}

//

//sum = sum + grade;

//

//y++;

// }

//

//if (excluded > 1) {

//System.out.printf("%s has been excluded at %d grade%n", name, y);

// } else {

//double avgGrade = sum / 12;

//System.out.printf("%s graduated. Average grade: %.2f%n", name, avgGrade);

4. Работа със Стринг/Чар:

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* charAt \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//definirame - chakame string - iskame daden simvol ot daden string

//char symbol = scanner.nextLine().charAt(0,1,2...etc)

//int num = symbol - mojem da zapazim stoinostta v int

//Scanner scanner = new Scanner(System.in);

//char symbol = scanner.nextLine().charAt({position of});

//int sum = symbol;

//System.out.println(num);

//

//String text = scanner.nextLine();

//

//for (int i = 0; i <text.length() ; i++) {

// char symbol = text.charAt(i);

// if (symbol < 'A' && symbol > 'Z')//not latin alphabet

//((symbol >= 'A' && symbol <= 'Z') || (symbol >= 'a' && symbol <= 'z'))

////chars from latin alphabet

//System.out.println(symbol); -> ignores non-alphabetic chars

// }

5. Намиране на Нечетно число:

// FIND ODD NUMBER for AMOUNT OF ITERATIONS

//int number = Integer.parseInt(scanner.nextLine());

//int sum = 0;

//

//

//for (int i = 1; i <= number; i++) {

////important formula for oddNumber for each iteration

//int oddNumber = 2 \* i - 1;

//System.out.println(oddNumber);

//sum += oddNumber;

//

//

//}

//System.out.printf("Sum: %d", sum);

6.Обръщане на Стринг:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*REVERSE STRING \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//String input = scanner.nextLine();

//String reverse = "";

//for (int position = input.length() - 1; position >= 0; position--) {

//

//char currentSymbol = input.charAt(position);

//reverse += currentSymbol;

////

7.Масиви/Arrays

Кога ни трябват масиви? Когато не мога да реша задачата без да работя с всички елементи наведнъж.

1.Array Declaration;

{data type} [] {name};

2.initialization of new array:

int[] intArray = new int[];

int[] intArray = new int[] {1,2,3,4,5};

3.Creating array and filling it through scanner:

Scanner scanner = new Scanner(System.in);

//заделяме число (големина на масива)

//int numbersCount = scanner.nextInt();

//int numbersCount = Integer.parseInt(Scanner.nextLine());

int[] numbers = new int[numbersCount];//инициализираме големината на масива

for(int i = 0; i < numbers.length; i++){

numbers[i] =scanner.nextInt();//пълним текущата позиция с число от конзолата

}

4.Четене отзад напред на масива:

for(int i = numbers.length - 1; i>=0; i--){

System.out.println(numbers[i]); //отпечатваме масива на обратно

}

5.Изпечатване на стойностите в масива преди инициализация:

for(int i = 0; i < numbers.length; i++){

Sout(numbers[i] + " ");

}

6.\*\*Алгоритъм\*\* за разделяна на елементите в масив със запетая:

for( int i = 0; i < numbers.length; i++){

System.out.print(numbers[i]);

if(i != numbers.length - 1){

System.out.print(",");

}

}

7.Четене от 1 ред в конзолата и разделяне на стойностите:

String lineOfNumbers = scanner.nextLine();

//вадим символи от стринга и ги обръщаме в числа

// [\\s+](file://s+) спейс ++

String[] numbersAsStrings = lineOfNumbers.split("\\s+");//разбиваме главния стринг по стрингове числа

int numbers = new int[numbersAsStrings.length];//взема големината на стринг масива

for ( int i = 0; i < numbers; i++){

numbers[i] = Integer.parseInt(numbersAsStrings[i]);

}

8.Сумиране директо на елементите от стринг масив:

String lineOfNumbers = scanner.nextLine();

String[] numbersAsStrings = lineOfNumbers.split("\\s+");

int sum=0;

for ( int i = 0; i < numbers; i++){

sum += Integer.parseInt(numbersAsStrings[i]);

}

System.out.println(sum);

9.Обръщане на стойностите на масив \*\*Алгоритъм\*\*

for(int i = 0; i < numbers.length / 2; i ++){

int otherIndex = numbers.length - 1 - i;//изчисляваме стойността на 2рия индекс

//разменяме 2те променливи:

int currentNUmber = numbers[i];//записваме стойността на i

numbers[i] = numbers[otherIndex];//слагаме новата стойност на i

numbers[otherIndex] = currentNUmber;//слагаме новата стойност на otherIndex

}

10.Четни/нечетни числа в масив (извличане алгоритъм)

int evenSum=0;

int oddSum=0;

for (String s : numbersAsStrings){

int number = Integer.parseInt(s);

if(number % 2 ==0){

evenSum += number;

}else{

oddSum += number;

}

+++++        ++++++++++++++++++++++++++

Правим масив в който набиваме четни и нечетни спрямо техния остатък {0,1}

int[] sums = new int[2];

for (String s : numbersAsStrings){

int number = Integer.parseInt(s);

sums[number % 2] += number; // добавяме четна/нечетна стойност директно в масива;

}

Sout(sums[0] - sums[1])

11.Сравняване на масиви \*\*\*Алгоритъм\*\*

boolean arraysAreEqual;

int differenceIndex = 0;

if(firstArray.length != secondArray.Length){

arraysAreEqual = false;

}else{

arraysAreEqual = true;

for(int i = 0; i < firstArray.length; i++){

if(firstArray[i] != secondArray[i]){

arraysAreEqual = false;

differenceIndex = i;

}

}

}

Sout(arraysAreEqual);

Ctrl + B - > отваря документацията

12.Събиране на стойностите на масив и кондензиране в друг масив.

// Боравене само с 1 масив и ченете на първия му елемент:

int[] numbers = new int[];

for int repetitions = 0; repetitions < numbers.length - 1; repetitions++){

for(int i = 0; i < numbers.length - 1; i++){

numbers[i]=numbers[i]+ numbers[i+1];

}

}

System.out.println(numbers[0]);

++++++++++ 2ND OPTION +++++++++++++++++++++++++++++++

//Създаваме втори масив и пълненето на елементи в него -1 елемент:

int[] numbers =

while(numbers.length > 1{ //проверяваме дължината на главния масив

//създаваме нов масив (дължина на оригиналния -1 за всяка итерация)

int[] condensed = new int[numbers.length - 1];

// i + 1 до предпоследния елемент

for(int i = 0; i < numbers.length - 1; i++){

//набиваме новата получена стойност в дадения индекс

condensed[i] = numbers[i] + numbers [i + 1];

}

//присвояваме новите стойности

numbers = condensed;

}

System.out.println(numbers[0]);

13.Четем стринг - записваме в инт масив:

String reveiced = scanner.nextLine();

//split създава сам по себе си масив, не е необходимо да го дефинираме предварително.

int firstNumber = Integer.parseInt(reveiced.split("\\s+")[0]);

String[] splittedArray= input.split("\\s+");

int firstNumber = Integer.parseInt(splittedArray[position])

+++++++Methods++++++++++++++

int number = new Scanner(System.in).nextInt();

methodname(number);

\*\*\*\*\*\*\*\*\*ArrayList\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Ctrl + P -показва какво се очаква като аргументи в метода

1.Declaration:

1.ArrayList/List <String/Integer/Double/Character..> name(stringList) = new ArrayList<String/Integer/Double..>();

//даваме начални параметри, можем да добавяме и махаме параметри в него:

2.ArrayList/List <..> name(stringList) = new ArrayList<..>(Arrays.asList(стойности));

// read a line, split into words and add them to a collection.

words.addAll(Arrays.asList(input.nextLine().split("\\s+")));

2.Print:

System.out.println(stringList.get());

3.Обхождане:

for(int i = 0; i < stringList.size();i++){ System.out.println(stringList.get(i))}

последен индекс = stringList.size() - 1;

4.Четене и добавяне на дума при дадено условие:

//Класически алгоритъм за четене на дума и обработване

Str word = scanner.nextLine;

While(!word.equals("")){

stringList.add(word);

word = scanner.nextLine

}

//Можем да принтираме с обикновен foreach:

for(String s : stringList){

System.out.println(s);

}

5.Премахване на елементи от листа:

//Никога не се махат с for цикъл, а с while

List<Integer> numbers = new ArrayList<Integer>();

numbers.remove(numbers.size()-1);->премахваме последния индекс;

numbers.remove(..) - премахва по индекс;

remove() - премахва по индекс;

remove(Integer.valueOf()) ->премахва по стойност

6.Методи на лист:

numbers.add(100);->добавя елемент в края на листа;

numbers.add(0, 100)->добавя стойност на даден индекс;

.size() - > вземаме размера на листа;

.get()-> вземаме елемент на дадена позиция;

.indexOf("abc") -> търсене на стойност по индекс;"-1" когато няма резултат;

.contains(element)->показва дали се съдържа;(indexOf is better)

6.Алгоритъм за намиране на елемент:

int index = stringList.indexOf("");

if (index == -1){

System.out.println("not found");

}else{

System.out.println("found")

}

7.Aлгоритъм за премахване на всики елементи с дадена стойност:

removeAll(List<String> elements, String value){

while(elements.remove(value)){

}

//Премахване на елементи при някакво условие:

int i = 0;

while(i < elements.size()){

if(elements.get(i).equals(value)){

elements.remove(i)

}else{

i++

}

}

//При листове които можем да обхождаме отзад напред:

int i = elements.size()-1;

while(i >=0){

if(element.get(i).equals(value)){

elements.remove(i);

}

i--;

}

}

8.Четене на числов ред от конзолата към List:

Alt+R+X+M ->Method extraction

List<Integer> numbers = new ArrayList<>("initial capacity - добро за оптимизация");

String lineOfNumbers = scanner.nextLine;

String[] numbersAsStrings = lineOfNumbers.split(""\\s+);

//Нормален for

for(int i = 0; i < numbersAsStrings.length;i++){

numbers.add(Integer.parseInt(numbersAsStrings[i]));

}

//foreach

for(String s: numbersAsStrings){

numbers.add(Integer.parseInt(s));

}

numbers.foreach(number ->System.out.print(number + " ")

9.Обхождане на лист с числа/сумиране Алгоритъм:

List<Integer>numbers = //четем от конзолата;

List<Integer> nextNumbers = sumAdjacentEqualNumbers;

while(nextNumbers.size != numbers.size()){

numbers = nextNumbers;

nextNumbers = sumAdjacentEqualNumbers(numbers);

}

private static List<Integer> sumAdjacentEqualNumbers(List<Integer> numbers){

int i = 0;

while(i < numbers.size()){

if(i < numbers.size() - 1 // от последния елемент){

nextNumbers.add(numbers.get(i) + numbers.get(i + 1);

i +=2;

}else{

nextNumbers.add(numbers.get(i));

i++;

}

}

}

10.Алгоритъм за филтриране на стойности по някаква критерия:

//когато искаме оригинала да е останал такъв какъвто си е.

private static List<Integer> filteredNumbers(List<Integer> numbers){

//най-лесния вариант е да създадем нов лист и да тъпчем стойности в него;

List<Integer> filteredNumbers = new ArrayList<>();

for( int number : numbers){

if(number >=0){

filteredNumbers.add(number);

}

}

return filteredNumbers

}

11.Четене на стринг от конзолата и записването му директно в лист:

List<String> numbersAsString = new ArrayList<>(Arrays.asList(scanner.nextLine().split("\\s+")));

List<Integer> numbers = new ArrayList<>();

numRowString.forEach(number->numbers.add(Integer.parseInt(number)));

++++++++++++++Associative Arrays++++++++++++++++

GET/PUT/REMOVE/ITERRATE -> най-важните стъпки за всички структури данни.

HashMap -> faster

LinkedHashMap -> връща записите по реда на добавяне

TreeMap-сортира по нарастване

1.Деклариране на Map

Map<Key, Value> {name} = new HashMap/LinkedHashMap/TreeMap<>();

2.Добавяне на елементи в Map:

Добавяме обект от тип студент с ключ 123.

{name}.put(123, new Student(facNum:123, name:"Ivo", age:28));

3.Какво е мапване?

-Преобразуването на елемент от 1 вид в друг, няма общо с асоциативните масиви,

използва се като фраза за преобразуване.

4.Принтиране на Map:

1.1 Принтиране чрез използване на стрийм.

countChars.entrySet().forEach(entry -> System.out.println(entry.getKey() + " - > " + entry.getValue()));

countChars.forEach(Character, Integer)->System.out.println(character + "->" + integer));

1.2 Принтиране чрез нормален Foreach

//for(var entry : countChars.entrySet())-съкратен запис

for (Map.Entry<Character, Integer> entry: countChars.entrySet()){

//обхождане и принтиране на мапа

System.out.println(entry.getKey() + " -> " + entry.getValue());

}

Sout({name}.get("key value" може да е стринг, номер и тн.));

-можем да принтираме през remove:

Sout({name}.remove(key:));->принтира стойността която е премахната;

5.Премахване на елементи:

{name}.remove(key:);

6.как да обходим:

Всеки мап в Java си има Множество от записи -> entrySet, което може да се итерира.

{name}.entrySet

for(Map.Entry<String, Integer>entry : {name}.entrySet()){

entry.getValue()->Стойност;

entry.getKey()-> дава ключа

}

Обхождаме записи от тип String - ако ключа е стринг, и Integer в случай на value от Integer.

Всеки един от елементите му като двойка <ключ, стойност>

7.Методи за съдържане:

Sout({name}.containsKey(key);

По-добър начин за проверка: \*Алгоритъм\*

Integer result = {name}.get("key");

if(result !=null){

Sout("yes" + result);

}else{

Sout("No")

}

8.Примерна задача за брой срещания на дадена стойност в Лист. Алгоритъм\*\*

List<Double> numbers = Arrays.<Double>asList(1.0, 2.0,3.0);

//Double е стойността, която търсим, а Integer е пътите срещания;

Map<Double, Integer> numberCounts = new TreeMap<>();

for(Double number : numbers){

Integer currentCount = numberCounts.get(number);

//долния код може да се замести с numberCounts.merge(number, 1, Integer::sum);

//map.merge(key, msg, String::concat)

if(currentCount != null){

numberCounts.put(number, currentCount + 1);

}else{

numberCounts.put(number, 1);

}

}

for(Map.Entry<Double, Integer> entry : numberCounts.entrySet()){

Sout(entry.getKey + ":" + entry.getValue());

}

9.Проверка думи по техните дължини:

Когато имаме 1 ключ срещу когото стои списък от данни

List<String> words = Arrays.asList("asd","hello","hi","yes");

Map<Integer, List<String>> wordsByLength = new TreeMap<>();

for(String word :words){

int key = word.length();//дължината на думата е ключовата стойност

//дай ми по тая дължина стойностите

List<String> wordsWithThisLength = wordsByLength.get(key);

//Map.computeIfAbsent Method - прави същото като проверката долу

if(wordsWithThisLength == null){//ако до сега не сме срещали такъв ключ/дължина

wordsWithThisLength = new ArrayList;//създаваме списъка

wordsByLength.put(key,wordsWithThisLength);

}

wordsWithThisLength.add(word);

}

for(Map.Entry<Integer, List<String>> entry : wordsByLength.entrySet()){

Sout(entry.getKey() + "->");//принтираме ключа на даденото ентри

for(String word : entry.getValue()){

//правим цикъл за обхождане на всички стойности

Sout(word + " ");

}

}

++++++++++++Lambda && Stream API ++++++++++++++++++++++

"->" (goes to) - показва ламбда операция;

{input parameters} -> {expression/statement}

Ламбдите са съкратен запис на функция.

1.Листи;

.filter()-всичко което можем да напишем в if, можем да го напишем и тук.Връща 1 или 0 елемента.

List<Integer> numbers = Arrays.asList(1,2,3,4);

List<Integer>filtered = numbers.stream()//обръщаме ги към поток от данни

.fillter(number -> number > 2 && number < 8)

.fillter(number -> number % 2 == 0)//filter проверява дали да добави, а map променя и добавя

.map(number -> number \* 2)//мап взема обект от някакъв тип и създава друг обект

.map(number-> "number:" + number)//обръщаме ги към стринг, но трябва да променим и листа на стринг

.collect(Collectors.toList());//обръщаме ги към лист отново

List<String> numbers = Arrays.asList("1","2","3");

List<Integer> result = numbers.stream()

.map(s->Integer.parseInt(s))

.map(n-> n \* 2)

.sorted((a,b) -> a.comparedTo(b))//сортираме ги а,както се подрежда с b(стандартната подретба), ако искаме да обърнем b.comparedTo(a)

.collect(Collectors.toList());

2.Разлика на стрийма м/у Масив и Лист:

-Масивите нямат numbers.stream() метод, ами е Arrays.stream(numbers)

int[] numbers = new int[]{1,2,3,4,5,6,7};

String[]strings = new String[]{"1","2","3"};

.map(s->Integer.parseInt(s))-> когато кажем map java разбира че ще създаваме обекти от Integer type, a не от примитивен тип int.

затова няма да работи при масиви, а работи при листи. При масивите трябва да е mapToInt.

int[] result = Arrays.stream(strings)

.mapToInt(s->Integer.parseInt(s))

.min()/.max()/.average() // могат да се използват само при масив mapToInt и тн.

.summaryStatistics //можем да го ползваме и него

.toArray();

string -> Integer.parseInt(s) , когато работим само с един параметър можем да извикаме

съкратен негов запис -> Integer::parseInt

List<Students> students = new ArrayList(Arrays.asList....);

\*New Student(facNum: 1243, name:Ivo, age:28)\*

OptionalInt minAge = students.stream()

////Намираме минималната възраст на студентите

.mapToInt(s->s.getAge())

.min();

///Ако искаме да запазим другите стойности зад да ги изведем, да вземем целия обект:

Object<Student> min = students.stream()

.min((a,b)-> Integer.compare(a.getAge(), b.getAge()));//Ако знаем този подход как работи не ни трябва mapToInt.

Student result = min.get();

3. Четене на елементи на масив и записването им в Лист:

String[] split = input.split("\\s+");

List < String> result = Arrays.stream(split)

.filter(s->s.length() % 2 == 0) // добавяме ако дължината на стринга е четна

.collect(Collectors.toList());

4. Обработване на Map<> с лист от обекти:

Map<String,List<Students>> obj = new HashMap<>();

for (Map.Entry<String, List<Students>> entry: obj.entrySet() ) {

// [0,1,2,3,4] -> < key1,value1>

//<key2,value2>

List<Students> studs= entry.getValue();//Списъка за текущия ключ

for (Students student: studs) {

System.out.println(studs.toString());

}

entry.getValue().forEach(User::toString);

}

5.Вземане на първия елемент от Лист колекция, ако съществува:

List.stream().filter(e-> e.getName().equals(objectName)).findFirst().orElse(null);

6.Вземаме определено поле от обект, който се намира в списък с обекти:

@Override

\*\*public Collection<Animal> getAnimals() { return animals; }\*\*

(animals.size() == 0 ? "none" : animals.stream().map(Animal::getName).collect(Collectors.joining(" ")))

7.Когато искаме да принтираме информацията на различни обекти и да ги върнем в стринг:

return areas.stream().map(Area::getInfo).collect(Collectors.joining(System.lineSeparator()));

double allkg = area.getAnimals().stream().mapToDouble(Animal::getKg).sum();

Вземане на стойност от обект и сумирането й

foods.stream().mapToInt(Food::getCalories).sum(); -> чрез използване на масив стрийм

foods.stream().map(Food::getCalories).reduce(0, Integer::sum); -> лист стрийм

Обхождане на Лист и филтрирането по даден критерий, търсим определен(първи обект) или връщаме NULL.

areas.stream().filter(a -> a.getName().equals(areaName)).findFirst().orElse(null);

Ако искаме да принтираме със запетая от Integer List примерно:

sout(numbers.stream().map(String::valueOf).collect(Collectors.joining(", "));

Намираме първия годен работник с повече от 50 енергия.

Helper helper = helperRepository.getModels().stream().filter(h->h.getEnergy() > 50).findFirst().orElse(null);

Намиране на счупените инструменти:

return instruments.stream().filter(Instrument::isBroken).count();

Вземаме първия елемент от Лист с даден Type на една абстракция.

return toys.stream().filter(t->t.getClass().getSimpleName().equals(type)).findFirst().orElse(null);

List<Explorer> suitableExplorers = explorerRepository.getCollection().stream().filter(e -> e.getEnergy() > 50).collect(Collectors.toList());

long retiredExplorers = suitableExplorers.stream().filter( e -> !e.canSearch()).count();

++++++++TEXT PROCESSING+++++++++++++++++++++

1. Алгоритъм за повтаряне на стринг:

//повторяемост при различни поредици/масиви/

String[] words = scanner.nextLine().split("\\s+")

//List<String> repeated = new ArrayList<>();

for(String word: words){

//repeated.add(repeat(word,word.length);

Sout(repeat(word, word.length()));

}

private static String repeat(String word, int repetitions){

char[] repeated = new char[word.length() \* repetitions];

for(i=0;i< repeated.length;i++){

repeated[i] = word.charAt(i % word.length());

}

return new String (repeated);

}

1.Методи

substring(int startIndex, int endIndex)

endIndex -не включително

- две вариации на substring:

substring(int start, int end)

substring(int start)-> от индекса до края

2. Алгоритъм за разбиване на задача по даден символ:

String text = "the quick brown fox jumps over the lazy dog";

char split = 'q';

int splitIndex = -1;

for( int i = 0; i < text.length(); i ++){

if(text.charAt(i) == split){

splitIndex = i;

break;

}

}

if(splitIndex != -1){

String beforeSplit = text.substring(0, splitIndex);

String afterSplit = text.substring(splitIndex);

Sout(beforeSplit);

Sout(afterSplit);

}else{

Sout(text);

}

3.IndexOf: -Aлгоритъм за намиране на всички индекси на нещо.

String text = "the quick brown fox jumps over the lazy dog";

char search = 't';

int index = text.IndexOf(search);

while(index != -1){

Sout(index);

index = text.indexOf(search, index +1);

//намираме всички индекси на нещо в даден низ

}

4.Премахване на определено нещо от текст;Постоянно започваме от нулата;Алгоритъм\*\*

String text = "the quick brown fox jumps over the lazy dog";

String removeWord = "fox"

int removeIndex = text.indexOf(removeWord);

while(removeIndex != -1){

String firstPart = text.substring(0,removeIndex);

String secondPart = text.substring(removeIndex + removeWord.length());

text = firstPart + secondPart;

}

5.replace(match,replacement)://заменяме всички срещания на даден низ

String[] bannedWords = new String[]{"the","fox"};

for(String bannedWord: bannedWords){

String censored = repeat("\*", bannedWord.length());

text = text.replace(bannedWord,censored);

}

++++++StringBuilder++++++++++++++++

//Ако трябва да вземаме символи и да ги добавяме някъде трябва да е в билдъра

//чак когато приключим с операциите го превръщаме в стринг -immutable;

StringBuilder stringBuilder = new StringBuilder();

stringBuilder.append();

String s = stringBuilder.toString();

stringBuilder.setCharAt();

6.Алгоритъм:

->генерален алгоритъм за решаване на проблеми, когато минаваме през нещо,

трупаме в буфер и накрая добавяме буфера в крайния резултат.

String text = "....";

char separator = '';

StringBuilder current = new StringBuilder();

List<String>split = new ArrayList<>();

for(i = 0; i < text.length();i++){

if(text.charAt(i) != separator){

current.append(text.charAt(i));

}else{

split.add(current.toString());

current.setLength(0);

}

+++++++++++REGEX+++++++++++++++++++++++++++

\b - граници на думи

\w - alphabet-цифри и \_ всеки 1 от символите по отделно, слагаме + ако искаме цели думи.

\W - всяко което не е символ на дума

\S -> всичко което не е white space

\d - digit от 0 до 9

\D -> всичко което не е цифра

[]-ако искаме да търсим който и да е от символите по отделно, групираме с [].

{от, до} -> интервал от, до, като втория аргумент не е задължителен

{3} точно 3 пъти, {3,} три или повече.

+ -> 1 или повече пъти;

\* -> 0 или повече пъти;

() - >обгръщаме целия израз като го правим един цял елемент, групираме

[^abc] -> '^' отрицание, в тези скоби показва какво не търсим

[abc^] -> ако ^ не е в началото е просто символа себе си.

? -> 0 или 1 път {0,1} върши същата работа

Именоване на групи:

(?<day>\d{1,2}-(?<month>[A-Z][a-z]{2})-(?<year>\d{4})) - 22-Jan-2015

(?:<day>\d{1,2}) -> ?: non capturing group не излиза в номерацията на групите

(?<month>Jan|Feb|March)-> | или символ

(?<year>\d{4})? ->годината не е задължителна

Когато обхождаме цял стринг и искаме той да е верен:

^regex$ -> ^ начало на текста, $ край на текста

\b\d{1,2}(<gr1>[/-])[A-Z][a-z]{2}\(gr)1\d{4}\b

Java&Regex:

Pattern pattern = Pattern.compile("a\*b";

Патърн е нещото което пишем регекса в него.

Matcher matcher = pattern.matcher("aaaab");

Нещото което се създава от патърна и търси по патърна.

1.Алгоритъм за създаване на патърн и намирането на всичко в текст.

Pattern pattern = Pattern.compile("regex");

Pattern pattern = Pattern.compile("regex", flags: Pattern.MULTILINE | Pattern.CASE\_INSENSITIVE);

String text = "...";

Matcher matcher = pattern.matcher(text);

1.1 фундаментално разбиране на matcher:

boolean foundSomething = matcher.find();

if(foundSomething){

Sout(matcher.group(0));

}else{

Sout("Not found");

}

++++++++++++++++++++++

boolean foundSomething = matcher.find(можем да сложим индекс);

while(foundSomething){

List<String>str..

Sout(matcher.goup(1,2.."groupName"));

str.add(matcher.group);

foundSomething = matcher.find();

}

2.replaceAll();

//text.replaceFirst();

String censored = text.replaceAll("regex", repl9acement:"\*\*\*\*\*\*");

3.text.matches -> не ни трябва начало и край на стринга"^..$", защото проверява

дали целия стринг който е подаден е съвместим.

///използва Comparator за сравняване на обекти в джава.

//{listName}.sort(Comparator.comparing({ClassName}::{getMethodName}));/

SortByAge-> persons.sort(Comparator.comparing(Person::getAge));

sortByName->persons.sort(Comparator.comparing(Person::getName));

/////////////////////////////////////////////////////////

STREAMS==>>>++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Java 8 offers a new answer: convert the entries into a stream, and use the comparator combinators from Map.Entry:

Stream<Map.Entry<K,V>> sorted =

map.entrySet().stream()

.sorted(Map.Entry.comparingByValue());

This will let you consume the entries sorted in ascending order of value. If you want descending value, simply reverse the comparator:

Stream<Map.Entry<K,V>> sorted =

map.entrySet().stream()

.sorted(Collections.reverseOrder(Map.Entry.comparingByValue()));

If the values are not comparable, you can pass an explicit comparator:

Stream<Map.Entry<K,V>> sorted =

map.entrySet().stream()

.sorted(Map.Entry.comparingByValue(comparator));

You can then proceed to use other stream operations to consume the data. For example, if you want the top 10 in a new map:

Map<K,V> topTen =

map.entrySet().stream()

.sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))

.limit(10)

.collect(Collectors.toMap(

Map.Entry::getKey, Map.Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));

Or print to System.out:

map.entrySet().stream()

.sorted(Map.Entry.comparingByValue())

.forEach(System.out::println);

++++++++++++++++++++++++++++++++++++++++++++++++++

MAP ITERRATE:

piecesInformation.forEach((key, value) -> System.out.printf("%s -> Composer: %s, Key: %s%n", value.getPieceName(), value.getComposer(), value.getKey()));

for (Map.Entry<String, Piece>entry: piecesInformation.entrySet()) {

System.out.printf("%s -> Composer: %s, Key: %s%n",entry.getValue().getPieceName(), entry.getValue().getComposer(), entry.getValue().getKey());

}

for (Map.Entry<String, Piece>entry: piecesInformation.entrySet()) {

Piece currentPiece = entry.getValue();

System.out.printf("%s -> Composer: %s, Key: %s%n",currentPiece.getPieceName(), currentPiece.getComposer(),currentPiece.getKey());

}

+++++++++++++++++++++++++++++++++++

Stack And Deque:

ArrayDeque<T> name = new ArrayDeque<>();

Stack -> push, pop, peek

Queue -> offer -> returns false,poll -> returns null if queue is empty,peek

while(stack.size() > 0)

while(stack.isEmpty())

for( Integer integer: stack)

PriorityQueue<T> queue = new PriorityQueue<>();

while(!queue.isEmpty()){

queue.poll(); //вземаме елемента с най-голям приоритет

}

stack.removeAll(Arrays.asList(69) -> премахва всички срещания, обаче трябва да се даде като колекция.

IntStream.rangeClosed(1,10).foreach(System.out:println);

Multidimentional arrays:

Прочитаме масив на 1 ред даден:

int[] arr = Arrays.stream(scanner.nextLine.split("\\s+")))

.mapToInt(Integer::parseInt)

.toArray();

Попълване на матрица:

int[][] matrix = new int[rows][]

for(int row = 0; row < rows; row++){

int[] arr = Arrays.stream(scanner.nextLine.split("\\s+")))

.mapToInt(Integer::parseInt)

.toArray();

//директно слагаме целия масив в дадения ред

firstMatrix[row] = arr;

}

Принтиране на матрицата:

private static void printMatrix(int[][] matrix){

for(int row = 0; row < matrix.length; row++){

//matrix.length -> броя на масивите

int[] arr = matrix[row];

for(int n: arr){

System.out.println( n + " ");

}

sout();

}

}

for(int row = 0; row < matrix.length; row++){

//matrix.length -> броя на масивите

int[] arr = matrix[row];

for(int col = 0; row < arr.length; col++){

System.out.println( arr[col] + " ");

}

sout();

}

}

for(int row = 0; row < matrix.length; row++){

//matrix.length -> броя на масивите

for(int col = 0; row < matrix[row].length; col++){

System.out.println( matrix[row][col] + " ");

}

sout();

}

}

for(int[] arr: matrix){

for(int n: arr){

sout(n + " ");

}

sout();

}

Четене на матрица:

private static int[][] readMatrix(Scanner scanner){

String[] dimensions = scanner.nextLine().split("\\s+");

int rows = ip(dimensions[0]);

int cols = ip(dimensions[1]);

int[][] matrix = new int[rows][cols];

fori(int row = 0; row < rows; row++){

int[]arr = Arrays.stream(scanner.nextLine().spilit("\\s+"))

.maptoInt(Integer::parseInt)

.toArray();

matrix[row] = arr;

}

return matrix;

}

Сравняване на матрици:

Arrays.deepEquals(firstMatrix, secondMatrix)

private static boolean compareMatrix(firstMatrix[], secondMatrix[]){

if(firstMatrix.length != secondMatrix.length){

return false;

}

for(int row = 0; row < firstMatrix.length; row++){

int[] firstArr = firstMatrix[row];

int[]secondArr = secondMatrix[row];

if(firstArr.length != secondArr.length){

return false;

}

for(int col = 0; col < firstArr.length; col++){

int firstElement = firstArr[col];

int secondElement = secondArr[col];

if(firstElement != secondElement){

return false;

}

}

return true;

}

Maps & Sets

Сета пази само уникални елементи/стойности

Set<E> set = new HashSet<>()/ TreeSet<>() / LinkedHashSet<>();

Set<Integer> set = Arrays.stream(scanner.nextLine.split("\\s+))

.map(Integer::parseInt)

.collect(Collectors.toCollection(LinkedHashSet::new));

Iterate and remove elements:

Iterator<Integer> firstIterator = set.iterator();

firstIterator.remove(); -> remove the element on the top (the first element)

firstIterator.next();-> provides with the element on the top;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*Streams and Files: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Стриймовете могат да бъдат използване и за четене и за писане.

Има 2 вида основни streams:

Входен: Input stream: когато идват данни от някъде.

Изходен Output stream: от апликацията излиза навън като изход от програмата.

Стриймовете са еднопосочни/входна или изходна/

Как да вземем файл и да създадем входен поток от него.

1.Намираме файла:

Четене на файла:

String path = "full/path/to/file"

FileInputStream fileInputStream = new FileInputStream(path);

String line = scanner.nextLine;

while(scanner.hasNextLine(){

System.out.println(line);

line = scanner.nextLine;

}

Четене по байтове:

read() е операция която връща следващия байт(character) под формата на ASCII стойност.

връща -1 ако няма повече.

try{

FileInputStream fileInputStream = new FileInputStream(path);

int bytte = fileInputStream.read()

while(bytte != -1){

System.out.print((char)bytte);

bytte = fileInputStream.read();

}

}catch (FileNotFoundException e){

System.out.println("File not found" + e.getMessage());

}catch (I0Exception e){

System.out.print("Cannot read from file")

}finally{

//не е задължителен блок

//можем да извършем операции които се изпълняват винаги накрая.

//главна причина освобождаваме ресурс

}

Try catch - могат да се комбинират много catch-ове,

но трябва да се подредят от генерализиран

по-конкретен, до по-общ.

----------------------

try {

FileInputStream inputStream = new FileInputStream(path);

int read = inputStream.read();

//четене на файл и принтиране на двоичния код.

while(read != -1){

System.out.print(Integer.toBinaryString(read) + " ");

read = inputStream.read();

}

}catch(I0Exception ignored){

//използваме името ignored за да игнорираме грешката

//I0Exception е по-генерална от FileNotFoundException,

//което може да избегне добавянето на 2ри catch:

//за четенето на файла, както и за ненамирането на файла.

}

try {

FileInputStream inputStream = new FileInputStream(path);

//създаване на файл чрез FileOutputStream с output-a,

//ако файла не съществува, изходния поток сам създава файла.

FileOutputStream outputStream = new FileOutputStream("output.txt");

int read = inputStream.read();

//четене на файл и принтиране на двоичния код.

while(read != -1){

//вземаме всяка дума по байтове като двоичен код и залепяме " "

String output = Integer.toBinaryString(read) + " ";

for( char c: output.toCharArray()){

//за всяко отпечатване в новия файл добавяме char-a от форматирания изход

outputStream.write(c);

}

//System.out.print(Integer.toBinaryString(read) + " ");

read = inputStream.read();

}

}catch(I0Exception ignored){

//използваме името ignored за да игнорираме грешката

//I0Exception е по-генерална от FileNotFoundException,

//което може да избегне добавянето на 2ри catch:

//за четенето на файла, както и за ненамирането на файла.

}

--------------------------------

Output stream to Integer instead of String(char)

FileInputStream inputStream = new FileInputStream(path);

FileOutputStream outputStream = new FileOutputStream("output.txt");

int read = inputStream.read();

PrintWriter writer = new PrintWriter()

while(read != -1) {

if(read == ' '){

writer.print(' ');

}else if(read == 10){

writer.print(' ');

}else{

writer.print(read);

}

read = inputStream.read();

}

//не трябва да забравяме да затворим Output стрийма;

writer.close(); ->вадим всичко от буфера и спираме стрийма.

writer.flush(); -> вадим всичко което е в буфера, без да прекратяваме стрийма.

writer.write(); -> пише във файл;

writer.println(scanner.next) -> докато има следващ елемент, by default SPACE.

----------------

Scanner reader = new Scanner(new FileInputStream(path));

------------------

PrintWriter - не е задължително да печата във файл.

FileWriter - идеята му е да печата във файл.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*Buffered Streams \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Четем определена информация, която се пази в буфер и вадим от него

докато не е празен - целта е по-малко излишни операции

BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(path)));

BufferedWriter writer = new BufferedWriter(new FileWriter(document-to-create.txt));

\*\*\*\*\*\*\*\*\*\*\*Functional programming Java\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Функцията не пази стойност, а действие което се извършва.

Функцията е обект, който пази информация как се извършва някакво действие, а не стойност.

Изолирана е от външния свят и не се влияе от него, винаги върши едно и също без промяна.

Stateless code -> unnamed function

{parameters} -> {body}

Arrays.stream(numbers).sum()

List<Integer> numbers = Arrays.stream(scanner.nextLine().split(", "))

.map(Integer::parseInt)

.filter(n -> n % 2 == 0)

//.sorted() //sorted(Comparator.reverseOrder())

.collect(Collectors.toList());

//print the values as string

String output = numbers.stream()

.map(String::valueOf)

.collect(Collectors.joining(", "));

System.out.println(output);

//use stream in the sout method

System.out.println(numbers.stream()

.sorted()

.map(String::valueOf)

.collect(Collectors.joining(", ")));

Функции обекти:

Function<[INPARAM], [OUTPARAM]> name = lambda expression x->x \* x;

int a = name.apply(inparam);

//same as void method -> print

Consumer<INPARAM> name = message -> sout

name.accept(inparam);

Supplier<OUTPARAM> name = () -> new Random().nextInt(maxVal);

int rnd = name.get();

Predicate<INPARAM> name = returns true or false {n -> n % 2 ==0;}

name.test(inparam)

UpperCaseWords:

Predicate<String> upperCaseWord = w -> Character.isUpperCase(w.charAt(0));

List<String> upperCase = Arrays.stream(scanner.nextLine().split("\\s+"))

.filter(upperCaseWord)

.collect(Collectors.toList());

System.out.println(upperCase.size());

System.out.println(String.join(System.lineSeparator(),upperCase));

UnaryOperator<PARAM> - функция която връща същия тип който е приела;

може да се ползва вместо Function с едни и същи оператори за вход/изход;

mapToInt/mapToDouble etc връщат примитивните стойности,

което е проблем при използване на функции и др, които изискват

Integer,Double и тн.

BiFunction<inA,inB,outC> name = (a,b) -> "Sum is " + (a + b);

\*\*\*\*\*\*\*\*\*\*\*\*DEFINING CLASSES\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Access modifiers of the fields:

private access only within the scope of the same class;

public - accessible from everyone;

protected -within the class / within the package / outside package by sunclasses

default - only within the same class and package;

Ако преизвикваме конструктор чрез друг конструктор, това

трябва да стане на първата операция задължително.

става чрез операцията this(params);

public constructor(params){

this(params..);

...

}

Статични методи ( статични клас членове)

общо поведение независимо от отделните обекти

извиква се с името на класа, точка и името на метода;

Integer.parseInt();

Math.pow();

Не трябва да имаме създаден нов обект за да ги използваме;

Зареждат се по същото време, когато се зареждат и класовете в паметта;

Static methods -> не са обвързани с обектите и не можем да извикваме референции

към полета "this.name"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reading Queue with Stream:

ArrayDeque<Integer> queue = Arrays.stream(scanner.nextLine().split("\\s+"))

.map(Integer::parseInt)

.collect(Collectors.toCollection(ArrayDeque::new);

ArrayDeque<Integer> steelInfo = new ArrayDeque<>();

Arrays.stream(scanner.nextLine().split("\\s+"))

.map(Integer::parseInt)

.forEach(steelInfo::offer);

------------------------------------------------------------

reading Stack with stream:

ArrayDeque<Integer> stack = new ArrayDeque<>();

Arrays.stream(scanner.nextLine().split("\\s+"))

.map(Integer::parseInt)

.foreach(stack::push);

Arrays.stream(scanner.nextLine().split("\\s+"))

.map(Integer::parseInt)

.forEach(stack::push);

----------------------------------

how to check if in Map all entrySet have positive values:

boolean moreThanZero = map.entrySet().stream().allMatch(e->e.getValue > 0);

-------------------

Print elements separated with ", " from Stack/ Queue

String remaining = queue.isEmpty() ? "none" : queue.stream

.map(String::valueOf)

.collect(Collectors.joining(", ");

sout("remaining: " + remaining);

-----------------------

Creating char Array from scanner and print it:

char[][] field = new char[size][size];

fori(...){

String line = scanner.nextLine();

field[i] =line.toCharArray();

if(line.contains("M"){

row = i;

col = line.indexOf("M");

}

}

for(char[] arr: field){

for(char c:arr){

sout(c);

}

sout();

}

----------------------------------------

Четене на матрица:

for (int row = 0; row < size; row++) {

String currLine = scanner.nextLine();

for (int col = 0; col < size; col++) {

char currChar = currLine.charAt(col);

armory[row][col] = currChar;

if (currChar == 'A') {

currPosition[0] = row;

currPosition[1] = col;

}

}

}

Jagged matrix:

char[][] map = new char[size][];

for (int r = 0; r < size; r++) {

String line = scanner.nextLine();

map[r] = line.toCharArray();

if (line.contains("P")){

row = r;

col = line.indexOf("P");

}

}

Създаване на абстрактен метод в клас Reader:

public class Reader {

Scanner scanner;

public Reader(InputStream inputStream){

this.scanner = new Scanner(InputStream);

}

public int[] readIntArray(String pattern){

return Arrays.stream(scanner.nextLine().split(pattern))

.mapToInt(Integer::parseInt)

.toArray();

}

Сравняване на стринг в иф проверка е по-добре да бъде написано по следния начин

String name = scanner.nextLine;

"String".equals(name){} //по този начин избягваме nullpointer exception

ако името/стринга е нул.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Abstraction/Encapsulation/Interfaces\*\*\*\*\*\*\*\*\*\*\*\*\*

+++++ Abstraction ++++++++++++++

How to achieve abstraction in java:

Methods, interfaces and Abstraction class.

Interfaces are 100% abstraction;

Abstract class 0% -100% abstraction -> between class (0% of abstraction)

and interface (100% of abstraction)

public class Person extents Mammal implements Animal

Mammal -abstract class;

Animal - interface;

+++ Abstraction -> процес на криене на имплементационните детайли,

показвайки само функционалност на потребителя; Постига се с

Абстрактни класове и интерфейси(също така методи, обикновени класове също);

+++ Encapsulation -> използва се за да скрие кода и данните в самостоятелно

звено за да защити данните от външния свят;Постига се с модификаторите за достъп

(private/protected/public/package private)

+++ Interfaces -> съдържат само и единствено сигнатура на публични методи,

нямаме право на други модифайъри

- задължава класа да иплементира методите, които са наследени от интерфейса,

или класа трябва да бъде дефиниран като абстрактен.

Можем да създаваме новите инстанции на класовете които го имплементират:

Interface - Animal

Class Dog implements Animal..

Class Cat implements Animal...

..psvm:

Animal dog = new Dog();//не е проблем да напишем и Dog dog = new Dog();

Animal cat = new Cat();//както и Cat cat = new Cat();

В референции от 1 и същ и тип можем да сложим различни имплиментации

на 1 и същ метод( примерно void eat()):

feedAnimal(dog);

feedAnimal(cat);

feedAnimal(new Animal() -> {

sout("Another implementation with with lambda..")

});

public static void feedAnimal(Animal animal){

//работи по различен начин в зависимост какво животно му подадем

animal.eat();

}

Когато имаме променлива в интерфейс, след компилация тя става:

public static final..., което не е много удачно.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class extends Class

Class implements Interface

Interface extends Interface

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

В един интерфейс методите по подразбиране са "public abstract.."

1 клас може да имплиментира множество интерфейси,

както и 1 интерфейс може да екстендва повече от 1 интерфейси.

Когато имаме метод от даден интерфейс, който трябва да се държи

по различен начин спрямо имплиментацията в която се намира,

не е удобно да използваме главен клас interfImpl, защото там трябва да

разпишем метода, но няма да има правилно поведение, когато изключим child

класовете му, по-добрия метод е да използваме абстрактен клас.

В абстрактните класове можем да имаме методи с иплиментация, така и

методи без иплиментация(абстрактни).

Абстрактните класове също както интерфейсите не могат да бъдат създавани

инстанции от тях.

Могат да задържат абстрактни методи, само когато са изрично дефинирани като

абстрактен клас и метода трябва да бъде дефиниран като абстрактен(abstract).

Не е задължително да добави иплиментация за всички наследени методи от интерфейсите.

Абстрактните методи не трябва да имат тяло с иплиментация.

\*\*\*\*\*\*\*Разлика м/у интерфейсите и абстрактните класове\*\*\*\*\*\*\*\*\*\*\*\*

Клас може да иплиментира много интерфейси, но само 1 абстрактен клас.

Интерфейсите не могат да имат модифайъри за достъп, всичко е public.

Абстрактния клас и интерфейса при добавяне на нов абстрактен метод,

задължават всички тяхни наследници да добавят иплиментация на този метод.

Когато се чудим дали да ползваме интерфейс или абстрактен клас ще трябва да попитаме:

"Ще има ли състояния класовете и обектите, които го иплиментират", ако ще добавяме състояния

и иплиментация на някакъв default метод, на нас ни трябва абстрактен клас.

List<Identifiable> identifiableList = new ArrayList<>();

String input = scanner.nextLine();

while(!"End".equals(input)){

String[] command = input.split(" ");

Identifiable identifiable = command.length == 2

? new Robot(command[1],command[0])

: new Citizen(command[0], Integer.parseInt(command[1]), command[2]);

identifiableList.add(identifiable);

input = scanner.nextLine();

}

String fakeId = scanner.nextLine();

identifiableList

.stream()

.map(Identifiable::getId)

.filter(i -> i.endsWith(fakeId))

.forEach(System.out::println);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\* Polymorphism \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Как проверяваме дали 1 обект е от даден тип:

цифрите в Java могат да се разделят с "\_", например 1\_000\_000.00

Car car = new Seat();

Нямаме идея какво ще бъде, но ни се налага да проверим какъв вид е информацията която идва:

public static void method(Car car){

//когато не сме сигурни какъв тип получаваме, а очакваме абстракция

//гъвкавия метод за проверка на вида на колата

if(car instanceof Seat){

Seat seat = (Seat)car;

sout("...%f",seat.getPrice());

}else if(car.getClass() = Audi.class){

Audi audi = (Audi)car;

sout("",audi.getPricePerDay());

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*GENERIC\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class Printer <T>{

private T thingToPrint;

public Printer(T thingToPrint){

this.thingToPrint = thingToPrint;

}

public void print(){

Sout(thingToPrint);

}

}

Когато го извикваме трябва да специфицираме какъв тип данни ще се пазят

Printer<Integer> printer = new Printer<>(23);

--------------

Generic метод, който сравнява елементи и намира минималния:

public static <T extends Comparable<T>> T min (List<T> list){

T min = list.get(0);

for(T element : list){

if(min.compareTo(element) > 0{

min = element;

}

}

return min;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Reflection\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Единствения начин да се инициализира тип, който има private конструктори

т.е не можем да го достъпим директно защото е скрит, този тип има 2 варианта:

1. никога не бива да бъде инициализиран - само статични методи в себе си;

2. класът може да се инициализира от статичен метод: Clas.<methods>;

Ако знаем името/типа на класа:

Class clazz = Dog.class;

Ако не знаем името на класа по време на компилация:

Class clazz = Class.forName("full package path HERE")

Class clazz = Reflection.class;

Class clazz = Class.forName("Reflection.Reflection");

-Правим инстанция на класа и го свързваме с рефлекшън класа - Class;

Reflection reflection = new Reflection();

Class clazz = reflection.getClass;

===

clazz.getName();

clazz.getSimpleName();

Class[] interfaces = clazz.getInterfaces();

Class className = clazz.getSuperclass();

Class[] interfaces = clazz.getInterfaces();

for (Class c: interfaces) {

System.out.println(c);

}

getInstance with Reflection:

1.вземаме конструктора:

Ако имаме няколко конструктора, подаваме параметрите

за конструктора който искаме да използваме.

Constructor constructor = clazz.getDeclaredConstructor(Тук можем да дадем параметри);

Object newInstance = constructor.newInstance();

Ако искаме новата инстанция да бъде от типа на Класа(Reflection в нашия случай),

от който ще извлечем информацията:

Class<Reflection> clazz = Reflection.class;

Class<?> superClass = clazz.getSuperClass();

Class<?>[] interfaces = clazz.getInstances();

for (Class<?> c: interfaces) {

System.out.println(c);

}

Constructor<Reflection> constructor = clazz.getDeclaredConstructor(Тук можем да дадем параметри);

Reflection newInstance = constructor.newInstance();

Field field = clazz.getField("somefield");

Field[] fields = clazz.getFields();

Field[] fields = clazz.getDeclaredFields();

String fieldName = field.getName();

Object fieldType = field.getType();

Function<Class<?>, String> formatType = c-> c == int.class ? "class int" : c.toString();

+++++++++++++++++++++++++++

class ReflectionUtils

public static <T extemds Member> Stream<T> filterMembersByName (T[] members, String filter){

return filterMembers(members, m-> m.getName().contains(filter));

}

public static <T extemds Member> Stream<T> filterMembers (T[] members, Predicate<T> predicate){

return filterMembers(Arrays.stream(members, predicate));

}

//overload

public static <T extemds Member> Stream<T> filterMembers (Stream<T> stream, Predicate<T> predicate){

return stream

.filter(predicate);

}

public static <T extends Member> TreeSet<T> collectByName(Stream<T> stream){

return stream.collect(Collectors.toCollection(

() -> new TreeSet<>(Comparator.comparing(Member::getName))));

}

++++++++++++++++++++++++++++

public class HighQualityCode

Class<Reflection> clazz = Reflection.class;

Method[] methods = clazz.getDeclaredMethods();

Field[] declaredFields = clazz.getDeclaredFields();

TreeSet<Field> fields = collectByName(Arrays.stream(declaredFields));

// ReflectionUtils must be imported as dependancy or used directly with ReflectionUtils.methodName

<ReflectionUtils.>filterMembers(fields.stream(), f -> !Modifier.isPriviate(f.getModifiers()))

.forEach(f -> System.out.println(f.getName() + " must be private!"));

TreeSet<Method> getters = collectByName(filterMembersByName(methods, "get"));

filterMembers(getters.stream(), g -> !Modifier.isPublic(g.getModifiers()))

.forEach(g -> System.out.println(g.getName() + " have to be public!"));

TreeSet<Method> setters = collectByName(filterMembersByName(methods, "set"));

filterMembers(setters.stream(), s -> !Modifier.isPublic(s.getModifiers()))

.forEach(s -> System.out.println(s.getName() + " have to be public!"));

\*\*\*\*\*\*\*\*\*\*\*\*\*\*ANOTATIONS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Присъстват по време на компилация и компилатора ги вижда,

могат да съдържат параметри.

На базата на тези параметри можем да правим нещо допълнително.

създаваме анотации като слагаме @ пред интерфейс клас.

слагат се 2 анотации отгоре над интерфейса

@Retention(RetentionPolicy.RUNTIME)-> къде ще се изпълнява анотацията:

-рънтайм

-сорс

-клас ( по време зареждането на класовете)

@Target(ElementType.METHOD) ->къде може да бъде поставяна тази анотация: Методи,Полета,Класове и тн.

Class<Reflection> clazz = Reflection.class;

Method method = clazz.getDeclaredMethod("setName", String.class);

Annotation[][] parameterAnnotations = method.getParameterAnnotations();

\*\*\*\*\*\*\*\*\*\*\* ERROR HANDLING \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Намират се в декларацията на класа,метода и се предават по веригата,

задължен е потребителя на <метода> да се справят с този exception.

Checked exceptions - Compile time - Base class is IOException

(FileNotFoundException, SocketException);

Unchecked exceptions - Runtime exceptions - base class is RuntimeException

(IllegalArgumentException, IndexOutOfBoundException, NumberFormatException);

Errors should not be handled, we should try to make our code clean and without throwing any

errors(OutOfMemoryError,StackOverflowError,LinkageError);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SOLID Principles \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*Design Patterns\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Има 3 вида патърни:

1/. Creational - Как да създадем и конфигурираме обекти,

които са по-комплексни за конфигурация и инициализация;

\* Ако започнем на много места да пишем New (нови депендансита,

които трябва да се съберат в 1 обект)

1:Singleton Design Pattern -> only one instance;

2:Prototype Pattern -> creating new instance of object A by method called

clone() literally copying the same object.

2/. Structural - по какъв начин трябва да ни е създадена структурата на кода,

как трябва да се подредени различни обекти, имплиментации и тн. като идеята е

да разреши проблеми които могат да възникнат към по-сложна и неясна структура.

\* Composition design pattern;

3/. Behavioral - поведенчески - кой какво ще върши, как са разпределени отговорностите,

как се разпределя делегацията м/у отделните класове, кой с кой ще комуникира, каква работа

ще се очаква и тн. \*Разделя и разпределя отговорностите на кода\*

Какво са дизайн патърните:

Абстрактна идея за разрешаване на даден проблем;

Не се интересуваме да имат 100% покритие на проблема с решението,

прилагаме решенията които помагат за нашия проблем(нагаждаме спрямо ситуацията);

Абстрактна идея/шаблон, използвана за решаване на различни проблеми,

когато характеристиките се препокриват до някаква степен с патърн шаблона.

Проблеми които решават дизайн патърните:

\*Abstraction;

\*Encapsulation;

\*Separation of concerns;

\*Coupling and cohesion;

\*Separation of interface and implementation;

\*Divide and conquer;

1.Singleton

public class Database{

private static Database instance;

private Database(){

//може да имаме инициализация на променливи или някаква друга логика,

//но не искаме да е създаването на инстанцията;

}

public static Database getInstance(){

//a.k.a lazy initialization -> докато не ми трябва, няма да го инициализирам;

if(instance == null){

//тук може да се случват някакви тежки операции;

instance = new Database();

}

return instance;

}

}

2.Facade

3.Command Pattern

4.Prototype Design Pattern

public interface Prototype<T>{

T clone();

}

public class Dog implements Prototype<Dog>{

...

@Override

public Dog clone(){

return new Dog(breed, name, birthday);

}

}

..

Dog dog = new Dog("husky", "pesho", 2022);

Dog dog2 = dog.clone();

5.Factory Design Pattern

6.Builder Design Pattern

7.Command Design Pattern

8.Strategy Design Pattern

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* JAVA DATABASE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

JDBC

ORM

Scanner scanner = new Scanner(System.in);

Properties properties = new Properties();

properties.setProperty("user","root");

properties.setProperty("password","1234");

Connection connection = DriverManager

.getConnection("jdbc:mysql://localhost:3306/diablo", properties);

Sout("Enter username: ");

String username = scanner.nextLine();

PreparedStatement query = connection.preparedStatement(

//винаги връща 1 ред, дори и той да е с инфо null заради JOIN-a

//        SELECT username, first\_name, last\_name, COUNT(ug.id) AS game\_count

//        FROM users

//        LEFT JOIN users\_games AS ug ON users.id = ug.user\_id

//        WHERE user\_name = ?);

//по-доброто решение за момента е вложена заявка, която не връща нищо, ако юзъра липсва:

SELECT user\_name, first\_name, last\_name

(SELECT COUNT(\*) FROM users\_games WHERE user\_id = u.id) AS games\_count

FROM users AS u

WHERE user\_name = ? AND last\_name = ?

query.setString(1, username);

query.setString(2, "Ivaylo");

ResultSet result = query.executeQuery();

if(result.next()){

//valid username

//user\_name, first\_name, last\_name, games\_count

String dbUsername = result.getString("user\_name");

String dbFirstName = result.getString("first\_name");

String dbLastName = result.getString("last\_name");

int dbGamesCount = result.getInt("games\_count");

souf("User: %s%n%s %s has played %d games", dbUsername, dbFirstName, dmLastName, dbGamesCount);

} else {

Sout("No such user exists")

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

enum Utils {

;

static Connection getSQLConnection() throws SQLException {

final Properties properties = new Properties();

properties.setProperty(Constants.USER\_KEY, Constants.USER\_VALUE);

properties.setProperty(Constants.PASSWORD\_KEY, Constants.PASSWORD\_VALUE);

return DriverManager.getConnection(Constants.JDBC\_URL, properties);

}

}

------------------------------------------------

enum Constants {

;

static final String USER\_KEY = "user";

static final String USER\_VALUE = "root";

static final String PASSWORD\_KEY = "password";

static final String PASSWORD\_VALUE = "root123";

static final String JDBC\_URL = "jdbc:mysql://localhost:3306/minions\_db";

static final String COLUMN\_LABEL\_NAME = "name";

}

------------------------------------------------

public class GetMinionNames {

private static final String GET\_MINIONS\_BY\_VILLAIN\_ID =

"select m.name, m.age from minions as m" +

" join minions\_villains mv on m.id = mv.minion\_id" +

" where mv.villain\_id = ?";

private static final String NO\_VILLAIN\_IN\_DB\_FORMAT = "No villain with ID %d exists in the database.";

private static final String GET\_VILLAIN\_NAME\_BY\_ID = "select v.name from villains as v where v.id = ?";

private static final String VILLAIN\_PRINT\_FORMAT = "Villain: %s%n";

private static final String COLUMN\_MINION\_AGE = "age";

private static final String PRINT\_FORMAT = "%d. %s %d%n";

public static void main(String[] args) throws SQLException {

final Connection connection = Utils.getSQLConnection();

final int villainId = new Scanner(System.in).nextInt();

final PreparedStatement villainStatement = connection.prepareStatement(GET\_VILLAIN\_NAME\_BY\_ID);

villainStatement.setInt(1, villainId);

final ResultSet villainResultSet = villainStatement.executeQuery();

if (!villainResultSet.next()) {

System.out.printf(NO\_VILLAIN\_IN\_DB\_FORMAT, villainId);

connection.close();

return;

}

final String villainName = villainResultSet.getString(Constants.COLUMN\_LABEL\_NAME);

System.out.printf(VILLAIN\_PRINT\_FORMAT, villainName);

final PreparedStatement minionsStatement = connection.prepareStatement(GET\_MINIONS\_BY\_VILLAIN\_ID);

minionsStatement.setInt(1, villainId);

final ResultSet minionsResultSet = minionsStatement.executeQuery();

//while (minionsResultSet.next()) {

for (int i = 1; minionsResultSet.next(); i++) {

final String minionName = minionsResultSet.getString(Constants.COLUMN\_LABEL\_NAME);

final int minionAge = minionsResultSet.getInt(COLUMN\_MINION\_AGE);

System.out.printf(PRINT\_FORMAT, i, minionName, minionAge);

}

connection.close();

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ORM FUNDAMENTALS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

JDBC позволява да си пишем сами заявките за по-прости и лесни за поддръжка приложения(грубия вариант на Java с SQL),

докато ORM надгражда това

Object-Relational Mapping:

Какво е ORM? -> Нещото което свързва приложението с базите данни.

Улеснява комуникацията между езика за програмиране и базата данни:

//Java classes -> Database Tables

//Database tables -> Java classes

Защо ни е да ползваме ORM?

\*\*\*\*\*\*\*\*\*\*\*HYBERNATE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Когато имаме вече конфигуриран Hybernete(ORM) с депендансита в Maven pom файла,

се нуждаем от няколко довършителни реда код за да се стартира:

psvm(){

Configuration configuration = new Configuration();

configuration.configure();//прочита XML-a и добавя всички настройки.

SessionFactory sessionFactory = configuration.buildSessionFactory();

Session session = sessionFactory.openSession();

session.beginTransaction();

Student fromDB = session.get(Student.class, 1);

Sout(fromDB.getId() + " " + fromDB.getName());

//session.getTransaction().comit();

//session.close();

}

\*\*\*\*\*JPA - Java/Jakarta(new) Persistance API \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

EntityManager entityManager = entityManagerFactory.createEntityManager();

entityManager.getTransaction().begin();

JPA дава интерфейси, които се онаследяват и развиват от Hybernate.

Пример за това е entityManager, entityManagerFactory интерфейсите който всъшност

са имплиментирани със (SessionImpl и SessionFactoryImpl)

отдолу в Hybernate..когато използваме JPA ние използваме тяхните интерфейси,

за да генерализираме и обобщим какво точно искаме да се случи, а имплиментацията

зависи от технологията която развива тези интерфейси.

Типичен пример е List интерфейса в Java който се имплиментира като ArrayList,LinkedList etc.

Стандарт за това какви операции искаме 1 ОРМ да поддържа

Групата интерфейси и анотации които определят как 1 ОРМ трябва да се държи

в рамките на Java за да бъде по спецификация.

Това което ни дава JPA са анотациите, и задължително разписани иплиментации на интерфейсите

които са задължителни за работата.В случая Hybernate дефинира как структурата наложена от JPA

ще работи и ще бъде имплиментирана.

Persistence -> конфигурира се в xml конфигурационен файл,

представлява като обект с информация необходима за дадена конфигурация.

1.JPA->\*import\* -> Persistence(createEntityManagerFactory())->EntityManagerFactory(createEntityManager())

->EntityManager/getTransaction();find();createQuery();persist();remove();/

2.Query/setParameter();getResultList();getSingleResult();executeUpdate();/

3.EntityTransaction/begin();commit();rollback();/

4.Entity:/id, field1,field2../->управлява се отново от JPA

Операции:

persist() => SQL INSERT

remove() => SQL DELETE by primary key

refresh() => SQL SELECT by primary key

detach() => премахва обекта от базата, но остава в променлива в Java кода;

merge() => update(синхронизира състоянията)

contains() => проверяваме дали нещо го има в (Persistence context)

flush() =>налей всичко в базата;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*Hybernate Code First Entity Relation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.Java persistence API Inheritance

2.Table Relations

Надграждаме с различни видове анотации и конфигурации:

1вия тип релация е наследяване:

Table per class strategy

@Entity

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

-> Поставя се на базовия клас, а не на конкретните класове.

TABLE\_PER\_CLASS отделен генератор за всеки клас

public abstract class Name

@Id

@GeneratedValue(strategy = GenerationType.TABLE)

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

Недостатъците са че се повтаря информация(полета) във всяко ентити(таблица)

Ако променим нещо в базовия клас, трябва да се промени във всеки наследник.

Няма foreign key-ове и таблиците не са свързани, третираме ги като индивидуални,

нямаме NULL стойности.

Генерира наследните Класове като таблици, без да генерира таблица за базовия клас(таблица)

генерира таблица "hibernate\_sequence" която се грижи за добавянето на нова информация

като следи ID-тата да не се повтарят измежду генерираните таблици.

\*\*GeneratedValue(strategy = GenerationType.TABLE)\*\*

Имаме няколко опции:

TABLE -> хибернер използва таблица за да генерира уникалното ИД на всеки обект измежду таблиците.

SEQUENCE ->

IDENTITY -> искаме базата данни да си определи какъв й е варианта на ауто инкремента и да си добавя стойности за ИДто.

2.Втората Стратегия е JOINED

По една таблица за всеки клас, но имаме и базова таблица за общите атрибути

Всяка една от таблиците трябва да има свой собствен PRIMARY Key

Базовия клас вече се генерира като таблица с връзки към наследниците:

@Entity

@Table(name = "vehicles")

@Inheritance(strategy = InheritanceType.JOINED)

public abstract class ...{

@Id

@GeneratedValue(Strategy = GenerationType.TABLE);

private int id;

Извличане от базата:

Car fromDB = entityManager.find(Car.class, 1L);

id = 1(L) защото е от тип long.

можем да използваме:

fromDB.getSeats() -> метод от самия клас.

fromDB.getModel() -> метод от базовия клас.

Разликата между @Basic и @Column е че и двете анотации са за дефиниране на поле,

но когато искаме да зададем стойност или да променим нещо по полето, трябва да използваме Column,

защото Basic не съдържа тези опции.

------------------------------

@Entity

@Table(name = "vehicles")

@Inheritance(strategy = InheritanceType.JOINED)

public abstract class ...{

@Id

@GeneratedValue(Strategy = GenerationType.TABLE);

private int id;

Ако използваме @Inheritance(strategy = InheritanceType.JOINED)

със @GeneratedValue(Strategy = GenerationType.IDENTITY);

Няма да имаме генерирана таблица за следене на ИД-тата,

по този начин във всяка таблица ИД-тата ще бъдат auto increment.

-----

В реални системи наследяването е повече от 2 класа, не винаги искаме таблици,

понякога има класове на които идеята им е просто да съдържат обща информация

която да се онаследява(като полета) без да се налага да създаваме таблица за тях.

В такива ситуации използваме

@MappedSuperclass -> инфото от тоя клас я сложи в следващата конкретна импрлиментация,

на наследника.

@Entity                                                -> @MappedSuperclass                                                                ->@Entity

Basic abstrac class Vehicle -> abstract class PassangerVehicle extends Vehicle -> class Car extends PassangerVehicle

public

Vehicle(String vehicleType){}->        PassangerVehicle(String CarType){super(CarType)}->public Car(){super(CAR\_TYPE)};

Ако пробваме онаследяването без анотацията @MappedSuperclass

всичко минава, но проблема идва от това че липсва информацията от мапнатия клас към

конкретния наследник клас, липсват полетата в таблицата;

JOINED стратегия overview:

Няма заявка която може да мине без JOIN, което го прави по-тежко.

Няма никакви НУЛ стойности, без повтаряща се информация,

имаме FOREIGN ключове.

3.Single\_Table Стратегия:

най-лесна като резултат, имаме само 1 цяла таблица,

и има най-доброто представяне откъм бързодействие

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "type")

Автоматично се създава Discriminator колона, която се грижи за определянето идентичността на

обекта, какъв вид точно е..Можем да определим името й чрез @DiscriminatorColumn(name = "type")

можем да определим също във всеки клас каква да бъде стойността, която го описва.

@Entity

@Table(name = "vehicles")

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "type")

public abstract Class Vehicle{

@Id

@GeneratedValue(strategy = GenerationType.TABLE)

private int id;

@Basic

@Column(insertable = false, updatable = false)

private String type; -> Ако сме казали изрично че искаме това да бъде

колоната която определя обекта, не трябва да бъде ъпдейтвана или инсъртвана.

Няма как клас Car да се превърне в клас Bike.

\*\*DataBase Relations in Java\*\*

1.One to One:

@Entity

@Table(name = "plate\_numbers")

class PlateNumber{

@Id

@GeneratedValue(strategy = GenerationType.IDENTIY)

private int id;

private String number;

@OneToOne(targetEntity = Truck.class, mappedBy = "plateNumber")

private Truck truck;

public PlateNumber(){}

}

@Entity

@Table(name = "has\_cars")

class Car{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

@OneToOne

@JoinColumn(name = "plate\_numbers\_id", referencedColumnName = "id")

//как ще се казва и с кое поле ще се свързва към другото ентити.

private PlateNumber plateNumber;//не може Entity да бъде съхранено като @Colum/@Basic

public Car(){}

}

Когато в един клас съдържаме данни от тип друг клас, връзката м/у тях е 1 към 1

За да кажем че 2 референции са 1 и също нещо между два класа можем да направим

следното:@OneToOne(targetEntity = Truck.class, mappedBy = "plateNumber")

Винаги имаме опцията да направим връзката само в едната посока, или в двете посоки.

Когато имаме нужда можем да го направим 2на дирекционна връзка.

Когато правим една връзка двойно дирекционна, винаги трябва да оказваме

къде се намира оригиналното дефиниране на тази връзка (targetEntity = Truck.class, mappedBy = "plateNumber")

за да може hybernate да знае че двете са 1 и съща връзка за да не създава

излишна 2ра такава.

2.One to Many vice versa

@Entity

@Table(name = "articles")

class Article{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String text;

@ManyToOne

@JoinColumn(name = "author\_id", referencedColumnName = "id")

private User author;

@ManyToMany(fetch = FetchType.EAGER)

@JoinTable(

name = "articles\_categories",

joinColumns = @JoinColumn(name = "article\_id", referencedColumnName = "id"),

inverseJoinColumns = @JoinColumn(name = "categories\_id", referencedColumnName = "id")

)

private Set<Category> categories;

public Article(){

this.categories = new HashSet<>();

}

public Article(String text){

this();

this.text = text;

}

}

@Entity

@Table(name = "users")

class User{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

@OneToMany(mappedBy = "author", targetEntity = Article.class,

fetch = FetchType.LAZY, cascade = CascadeType.ALL)

private List/Set<Article> articles;

public User(){

this.articles = new ArrayList<>();

}

public User(String name){

this();//викаме празния конструктор за да се инициализира списъка.

this.name = name;

}

}

@OneToMany/@ManyToOne съдържат опциите:

1.fetch - връща обекти от базата данни.

Ентититата които извлича се запазват в Persistence Context като cache.

когато Fetch-a е EAGER, то винаги ще се изпълни и извлече от базата.

Когато е LAZY, ще се извлече, когато допълнително му кажем get...

@ManyToOne -> by default всеки път се взема цялата информация. -> EAGER

@OneToMany -> по подразбиране не се взема цялата информация -> LAZY.

fetch = FetchType.EAGER -> искаме взеки път да вземаме всички статии които потребителя е написал

FetchType.LAZY ->

(ако вземем User, няма да вземем всички негови статии)

2.cascade = CascadeType.ALL/.DETACH/.MERGE/.PERSIST/.REFRESH/.REMOVE

дали да се изпълни дадена операция към дадения обект/обекти.

3.Many to Many

Hibernate създава мапинг раблица за да реализира връзката @ManyToMany.

За да конфигурираме цялата мапинг таблица използваме JoinTable.

@JoinTable(

name = "articles\_categories",

joinColumns = @JoinColumn(name = "article\_id", referencedColumnName = "id"),

inverseJoinColumns = @JoinColumn(name = "categories\_id", referencedColumnName = "id")

)

@Entity

@Table(name = "categories")

class Category{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

private String name;

@ManyToMany(targetEntity = Article.class, mappedBy = "categories")

private Set<Article> articles;

public Categories(){}

public Categories(String name){

this.name = name;

}

}

4.Self Referencing

\*\*\*\*\*\*\*\*\*\*\*\*SPRING INFO\*\*\*\*\*\*\*\*\*\*\*\*\*

1.Repositories: добавяме още 1 ниво на абстракция.

Идеята му е да имаме единица еднаква за всички ентитита които я ползват,

тя отговаря за CRUD операциите, автоматично генерира JPQL/SQL код,

като ни позволява да добавяме нащи къстъм операции или заявки в зависимост

какви са нашите нужди.

-Дава ни всичко което работи с базата.

Това което ни предоставя по подразбиране 1 JPA репозитори са операциите:

-save(S var1) -> на един елемент

-save (Iterable<S> var1) -> запазва колекция;

-T findOne(ID var1) ->да намерим ентити по ID;

-boolean exists(ID var1) ->;

-Iterable<T> findAll() -> връща колекция;

-long count();

-void delete(ID var1);

-void deleteAll();

Репозиторитата вече ще отговарят за 1 клас.

Имайки такова репозитори, можем да генерираме заявки на базата на

имената на методите.

@Repository

public interface StudentRepository extends

JpaRepository<Student,Long>, като дженерика е <TypeOfObject, primaryKeyType>{

List<Student> findByMajor(Major major);

}

като ще се генерира автоматично заявката според името на метода.

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

2.Spring Data Services:

Сървис нивата са местата където да си организираме бизнес логиката,

спринг използва патърн в който изграждаме логиката си в някакви слоеве,

трябва да имаме модели(класове), модел репозиторита, сървиси,

като идеята е да изолираме вътрешните операции.

При сървисите отново следваме това да имаме интерфейси и имплементации.

Цялата бизнес логика.

public interface StudentService {

void register(Student student);

void expel(Student student);

void expel(long id);

Student findStudent(long id);

List<Student> findStudentByMajor(Major major);

}

Интерфейса ни показва какви операции трябва да съдържа имплиментацията на сървиса:

@Service

public class StudentServiceImpl implements StudentService{

@Autowired

private StudentRepository studentRepository;

@Override

public void register(Student student){

studentRepository.save(student);

}

@Override

public void expel(Student student){

studentRepository.delete(student);

}

}

В рамките на нашия сървис добавяме репозиторито за определения клас

и в методите можем да използваме самото репозитори и неговите методи

върху обектите с които работим.

@Autowired -> казва на на спринг, когато създаваш обект от тип StudentServiceImpl,

намери ми studentRepository обект с който да мога да работя.Тоест

казваме на спринг да inject-не валидна инстанция на studentRepository

с която да може да се работи.

@Service анотацията показва че StudentServiceImpl е сървис,

използваме тези анотации защото по този начин можем да използваме

dependency injection-a на Spring, показваме че това е клас,

който Spring ще управлява.

По този начин казваме: давам ти този клас да го управляваш

и ми дай 1 инстанция на StudentRepository, която аз да мога да използвам

3. MainApplication.java

@SpringBootApplication

public class MainApplication{

public static void main(String[] args){

SpringBootApplication.run(MainApplication.class,args);

}

}

Това е главният клас, който ще запали нашето приложение.

4.Нашият PSVM метод вече се превръща в ConsoleRunner:

@Component-> за да знае спринг че трябва да управлява този клас.

public class ConsoleRunner implements CommandLineRunner{

@Autowired//подаваме интерфейс

private StudentService studentService;

@Autowired//подаваме интерфейс

private MajorService majorService

@Override

public void run(String...strings) throws Exception{

Major major = new Major("Java DB Fundamentals");

Student student = new Student("John", new Date(), major);

majorService.create(major);

studentService.register(student);

}

}

\*\*\*\*\*\*\*\*\* SPRING ADVANCED QUERIES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

JPQL -> SELECT/UPDATE/DELETE

"SELECT i FROM Ingredient AS i WHERE i.name IN :names"

:names -> така се изписват параметрите.

работим с класовете(Ingredient) и техните полета(name)

SELECT c

FROM <Class> AS c

JOIN c.<fieldName> AS fn

WHERE fn.<field> < :field

UPDATE:

UPDATE <Class> AS c

SET c.<priceVariable> = c.<priceVariable> \* 1.10

WHERE c.<nameVariable> IN :names

DELETE:

DELETE FROM <Class> as c

WHERE c.<nameVariable> = :name

Когато искаме да направим Custom заявки, ние имаме специално решение по което

трябва да се случи това:

1. Името вече не е толкова важно за да го пишем по спецификация на Спринг.

Трябва да използваме анотацията " @Query" над метода в който ще дефинираме заявката.

@Query("SELECT s FROM Shampoo as s " +

" JOIN s.ingredients as i" +

" WHERE i.name IN :ingredientNames")

List<Shampoo> findByIngredientsNames(Set<String> ingredientNames);

Тъй като имената са еднакви "ingredientNames" нямаше нужда да използваме допълнителната

анотация @Param, която може да се използва по следния начин при

разминаващи се имена.

@Query("SELECT s FROM Shampoo as s " +

" JOIN s.ingredients as i" +

" WHERE i.name IN :ingredientNames")

List<Shampoo> findByIngredientsNames( @Param(value = "ingredientNames") Set<String> ingredients);

\*\*\*\* ADVANCED REPOSITORIES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Можем да имаме главно репозитори и негови наследници.

Тъй като се екстендва JpaRepository и искаме Спринг да не го използва

директно, ние трябва да използваме анотацията @NoRepositoryBean.

Така казваме на Спринг, че това е нещо междинно и няма нужда да се

създава цяла отделна инстанция или да се опитва да създава.

@NoRepositoryBean

public interface IngredientRepository<T extends Ingredient>

extends JpaRepository<T, Long>{}

@Repository

public interface ChemicalIngredientRepository extends IngredientRepository

<BasicChemicalIngredient>{

LIst<ChemicalIngredient> findByChemicalFormula(String chemical);

}

3. Анотацията @Transactional, използваме я когато кода и заявките,

които са изписани в рамките на дефинирания под нея метод искаме да се случат в

една транзакция, когато имаме да направим повече операции, които искаме

да се извършат заедно.

3.1 Другата опция когато ще ни е необходимо да използваме @Transactional анотацията е

когато имаме LAZY fetching при run метода, за да не затваря предварително

комуникацията с базите, като това може да доведе то невъзможността за изпълнение на подзаявки и тн.

\*\*\*Custom Repository Creation \*\*\*\*\*\*\*

public interface CustomRepository{

void create(Object object);

}

-------------

@Repository

public class CustomRepositoryImpl implements CustomRepository{

@PersistenceContext

private EntityManager entityManager;

@Transactional

public void create(Object object){

entityManager.persist(Object);

}

}

\*\*\*\*\*\*\*\*\*Custom Spring Configuration \*\*\*\*\*\*\*\*\*\*\*\*\*\*

До момента това което сме използвали е application.properties файла

в който задаваме различни конфигурации, библиотеки и тн.

Можем да направим много сходно нещо чрез Java Class с необходимите анотации

и типове данни.

@Configuration -> оказваме на Спринг, че в този клас ще се опитаме да конфигурираме

някакви неща.

@EnableJpaRepositories(basePackage = "com.demo.demo") -> искам JPA репозиторита,

провери го в пакета "com.demo.demo", всички класове които съм изписал

стоят в този пакет.

@EnableTransactionManagement -> искаме да имаме транзакшън мениджмънт.

@PropertySource(value = "application.properties" -> със сигурност ще ни

трябват част от стойностите тип JDBC стринга, ще трябва да се извлекат и използват

от някъде. По този начин задаваме на Спринг в кой файл да търси.

@Query("UPDATE Ingredient i SET i.price = i.price + i.price \* :multiplier")

@Modifying -> идентифицира, че метода носи със себе си заявка от следния тип: INSERT/UPDATE/DELETE

и модифицира със себе си базата.

void increasePriceByPercent(@Param("multiplier")BigDecimal percent)

Ако някъде ни пристигат данни, които не са от точните и очаквани типове,

можем да ги преобразуваме в Service-ите

@Override

@Transactional -> добавяме го ако заявката е UPDATE/DELETE/INSERT както по-горния пример е свързан.

public void increasePriceByPercentage(double percent){

BigDecimal actualPercent = BigDecimal.valueOf(percent);

this.ingredientRepository.increasePriceByPercent(actualPercent);

}

-------------------------------------------------

\*\*\*\*\*\*\*\* Spring Data Auto Mapping Objects \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ДТО/DTO -> Дата трансфер Обджект

отговарят за предаване на опростени данни м/у backend/DB,

там където обработваме данните в приложението.

Като идеята им е да са контейнери с данни(само полетата) без да

имат в себе си някакви методи и тн.

по-простите класове(а.к.а домейн обекти) без много релации и информации могат също да бъдат използвани като ДТО.

м/у сървисите и уеб layer-a комуникацията става чрез тези ДТО-та.

Имайки много категории и релации в даден обект има възможност да се извлече твърде много излишна

информация от базата.

DAO -> Data Access Object -> служи за access към базата данни(много сходно до Репозитори)

ModelMapper -> преобразува един обект в друг обект.

//Converter служи за това да можем да кажем на модел мапъра как да преобразува един

базов тип в друг.

List<Employee> all = this.employeeService.findAll();

ModelMapper mapper = new ModelMapper();

TypeMap<Employee, CustomDTO> employeeToCustom = mapper.createTypeMap(Employee.class, CustomDTO.class);

Converter<Employee, Integer> getLastNameLength =

e -> e.getSource() == null ? null : e.getSource().getLastName().length();

//e.getSource() -> това е входната информация която искаме да мапнем.

employeeToCustom.addMappings(mapping ->

mapping.when(Objects::nonNull).using(getLastNameLength)

.map(Employee::getManager, CustomDTO::setManagerLastNameLength));

all.stream().map(employeeToCustom::map).forEach(System.out::println);

------------------------

ModelMapper mapper = new ModelMapper();

//създаваме конвъртър на който му показваме как да конвертира от един тип в друг.

Converter<String, LocalDate> localDateConverter = context -> LocalDate.parse(context.getSource());

//добавяме конвъртъра за да знае мапъра как да работи с дадените типове.

mapper.addConverter(localDateConverter, String.class, LocalDate.class);

Birthday birthday = new Birthday();

birthday.setDate(LocalDate.now());

BirthdayDTO dto = mapper.map(birthday, BirthdayDTO.class);

System.out.println(dto.getDate());

Birthday map = mapper.map(dto, Birthday.class);

System.out.println(map.getDate());

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* JSON Processing \*\*\*\*\*\*\*\*\*\*\*\*

JSON -> JavaScript Object Notation.

Библиотека GSON която се използва за работа с JSON.

Анотацията която ще използваме в тази библиотека е @Expose.

Ако едно поле няма @Expose анотацията, можем да използваме

excludeFieldsWithoutExposeAnnotation() което прескача директно полето.

Нещата които ще парсваме от и до JSON са DTO обектите.

Променя се частта как данните пристигат в нашето приложение и как ги изпращаме.

-setPrettyPrinting() -> чрез този метод форматираме самия JSON да е по-прегледен.

Gson gson = new GsonBuilder();

AddressJsonDTO addressJsonDTO = new AddressJsonDTO();

addressJsonDTO.setCountry("Bulgaria");

addressJsonDTO.setCity("Sofia");

addressJsonDTO.setStreet("Mladost 4");

String content = this.gson.toJson(addressJsonDTO);

-------------------------------

List<AddressJsonDTO> addressJsonDTOs = new ArrayList<>();

addressJsonDTOs.add(addressJsonDTOBulgaria);

addressJsonDTOs.add(addressJsonDTOSpain);

String content = this.gson.toJson(addressJsonDTOs);

-> получаваме масив с 2 обекта.

Хубаво е нашите сървиси да приемат и предават DTO-та.

Ако искаме да прочетем от някакъв файл:

AddressJsonDTO addressJsonDTO =

//(на това място се намира файла, такъв искам да бъде резултата)

this.gson.fromJson("/files/input/json/address.json",AddressJsonDTO.class);

По същия начин можем да накараме да ни даде масив с информация:

AddressJsonDTO[] addressJsonDTO =

//(на това място се намира файла, такъв искам да бъде резултата)

this.gson.fromJson("/files/input/json/address.json",AddressJsonDTO[].class);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Read File with JSON and Parse it to DB \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

FileReader fileReader = new FileReader(USERS\_JSON\_PATH); //path to file

//saving all Json to DTO object array

UserImportDTO[] userImportDTOS = this.gson.fromJson(fileReader, UserImportDTO[].class);

//parse all dtos to real object

List<User> users = Arrays.stream(userImportDTOS)

.map(importDTO -> this.modelMapper.map(importDTO, User.class))

.collect(Collectors.toList());

//save in db

this.userRepository.saveAllAndFlush(users);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Read JSON and set Random options to it \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public void seedProducts() throws FileNotFoundException {

FileReader fileReader = new FileReader(PRODUCTS\_JSON\_PATH);

ProductImportDTO[] productImportDTOS = this.gson.fromJson(fileReader, ProductImportDTO[].class);

List<Product> products = Arrays.stream(productImportDTOS)

.map(importDTO -> this.modelMapper.map(importDTO, Product.class))

.map(this::setRandomSeller)

.map(this::setRandomBuyer)

.map(this::setRandomCategories)

.collect(Collectors.toList());

this.productRepository.saveAllAndFlush(products);

}

private Product setRandomSeller(Product product){

Optional<User> seller = getRandomUser();

product.setSeller(seller.get());

return product;

}

private Product setRandomBuyer(Product product) {

if (product.getPrice().compareTo(BigDecimal.valueOf(900)) > 0){

return product;

}

Optional<User> buyer = getRandomUser();

product.setBuyer(buyer.get());

return product;

}

private Optional<User> getRandomUser() {

long usersCount = this.userRepository.count();

long randomUserId = new Random().nextLong(usersCount) + 1;

Optional<User> user = this.userRepository.findById(randomUserId);

return user;

}

private Product setRandomCategories(Product product) {

Random random = new Random();

long categoriesDBCount = this.categoryRepository.count();

long count = random.nextLong(categoriesDBCount);

Set<Category> categories = new HashSet<>();

for (int i = 0; i < count; i++) {

long randomId = random.nextLong(categoriesDBCount) + 1;

Optional<Category> randomCategory = this.categoryRepository.findById(randomId);

categories.add(randomCategory.get());

}

product.setCategories(categories);

return product;

}

Понеже в горните методи се случва да има дупликации между продукт и категория

ключовете задачата се чупи с exception duplicate primary key...

за да избегнем това ние трябва да имплиментираме следните 2 метода,

които автоматично се създават чрез инсърт в Категория обекта.

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Category category = (Category) o;

return Objects.equals(id, category.id);

}

@Override

public int hashCode() {

return Objects.hash(id);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*Принтиране на нестнати/вложени ДТО-та \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In CommandRunner:

List<UserWithSoldProductsDTO> allUsersWithSoldProducts =

this.userService.getAllUsersWithSoldProducts();

----------------------------

In UserServiceImpl:

@Override

@Transactional

public List<UserWithSoldProductsDTO> getAllUsersWithSoldProducts() {

List<User> allWithSoldProducts = this.userRepository.findAllWithSoldProducts();

-----------------------------------------------

IN userRepository:

@Query("SELECT u from User u " +

"JOIN u.sellingItems p " +

"WHERE p.buyer IS NOT NULL")

List<User> findAllWithSoldProducts();

---------------------------------------------

List<UserWithSoldProductsDTO> users = allWithSoldProducts

.stream()

.map(user -> this.mapper.map(user, UserWithSoldProductsDTO.class))

.collect(Collectors.toList());

return users;

}

----------------------------

String json = this.gson.toJson(allUsersWithSoldProducts);

System.out.println(json);

\*\*\*\*\*\*\*\*\*\*\*\*\*\* XML PROCESSING \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

За да работим с XML в Java ще използваме JAXB.

Marshalling - Java Object to XML;

Unmarshalling - XML to Java Object;

Трябва да анотираме Java обекта за да предоставим инструкции за създаването на XML-a:

@XmlRootElement(name = "address")

@XmlAccessorType(XmlAccessType.FIELD)

public class AddressDto implements Serializable{

@XmlAttribute(name = "country")

private String country;

@XmlElement(name = "city")

private String city;

}

XmlAccessorType.FIELD/

PROPERTY/

PUBLIC\_MEMBER;

XmlAttribute -> казва че дадено поле трябва да бъде съхранено.

XmlElement -> казва че дадено поле трябва да бъде прочетено.

XmlElementWrapper(name = "") -> ако колекция от елементи се нуждаем от тази анотация,

за да окажем какъв да бъде външния елемент ( името на ограждащия елемент)

XmlTransient -> ако искаме някой елемент да не бъде експортнат/импортнат

Когато започваме да работим с инстанция на JAXB трябва да си създадем един

JAXBContext за да можем да манипулираме обектите с операциите които

са заложени от библиотеката.

this.jaxbContext = JAXBContext.newInstance(Object.getClass);

Marshaller marshaller = context.createMarshaller();

marskaler.marshal()

как работим със списъци:

@XmlRootElement(name = "address")

@XmlAccessorType(XmlAccessType.NONE)

class AddressXmlDTO {

@XmlElementWrapper(name = "countries")

private List<CountryXmlDTO> countries;

@XmlAttribute(name = "vgdareno pole")

private Pole pole;

@XmlElement

private Pole pole2; -> заобграден от тагове

public AddressXmlDTO(int id, CountryXmlDTO country, String city){

..

this.country = List.of(country, country, country);

}

}

------------------------------

Main(){

CountryXmlDTO countryDto = new CountryXmlDTO("Bulgaria");

AddressXmlDTO xmlDto = new AddressXmlDto(5,countryDto, "Vidin");

JAXBContext context = JAXBContext.newInstance(AddressXmlDto.class);

Marshaller marshaller = context.createMarshaller();

//        marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);

marshaller.marshal(xmlDto, System.out);

\*\*\*\*\*\*\*\*\* Когато четем от конзолата XML -> Java Object \*\*\*\*

Unmarshaller unmarshaller = context.createUnmarshaller();

Object unmarshal = unmarshaller.unmarshal(System.in);

-> връща обект и трябва да го кастнем за да получим конкретен обект:

// AddressXmlDTO unmarshal = (AddressXmlDTO) unmarshaller.unmarshal(System.in);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Spring Fundamentals \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Главния клас на приложението за Спринг:

@SpringBootApplication

public class ExampleApplication{

SpringApplication.run(ExampleApplication.class, args);

}

Spring Boot Starters -> комбинирана група от депендансита, която е default-на от

спринг разработчиците идваща като самостоятелно депенданси.

Spring Boot Auto-Configuration -> начин да се скрие конфигурацията за да бъде

по-просто и лесно. ( @Bean/ @Component )

Spring Boot Actuator -> механизъм, който предоставя енд пойнти (за да можем по-лесно да наблюдаваме

нашия АП и какво се случва с него)

//Gradle

dependencies {

compileOnly("org.springframework.boot:spring-boot-starter-actuator")

}

//localhost:8080/health -> end point

//предоставя някакви енд пойни автоматично към http -> полезно за тест към браузър, какво излиза като информация;

Spring Data -> начин за да се улесни достъпа до различните видове DB системи,

без да използваме нещо специфично за определена DB.

2. Spring provides Inversion of Control and Dependency Injection;

Inversion of control е патърн който е заложен от спринг фреймуорка, който ни дава

насоки които да следваме за да се изпълни програмата, какво трябва да имплиментираме и тн.

Имплиментацията на патърна с Dependency injection:

@Service

public class UserServiceImpl implements UserServise{

@Autowired

private UserRepository userRepository;

}

-> нашия клас зависи от другите класове, като ние почти никъде не използваме оператора New,

като самия фреймуорк знае как да създаде и inject-не самите класове, които са записани

като необходими за работата на дадения клас.

1.Когато работим с дадения поток ние не трябва да знаем детайли за това каква определена

имплиментация ще дойде, защото не е работа на UserServiceImpl да знае с какво точно работи,

като той работи просто с интерфейса и не е необходимо да съдържа инициализация на входните

зависими класове.

-> Класовете които се инжектират се наричат "Beans".

Автоматичните бийнове се дефинират чрез анотациите: @Component, @Service, @Repository

като имаме експлицитна анотация за бийн: @Bean

//Spring IoC Container is the core of Spring Framework.

//It creates the objects, configures and assembles their dependencies, manages their entire life cycle.

//The Container uses Dependency Injection(DI) to manage the components that make up the application

Пример:

public interface Animal{

void makeNoise();

}

----

//@Component -> ако използваме тази анотация, няма да има нужда да слагаме @Bean анотацията

//в ZooCoanfig за този клас.

public class Dog implements Animal, BeanNameAware{

private boolean superDog;

public Dog(){

this(false);

}

public Dog(boolean superDog){

this.superDog = superDog;

}

@Override

public void makeNoise(){

if(superDog){

Sout("super bark super bark");

}else{

Sout("bark bark");

}

}

@Override

public void setBeanName(String name){

Sout("The name of this dog Bean is " + name);

}

}

---------------

public class Cat implements Animal{

@Override

public void makeNoise(){

Sout("meow meow");

}

}

---------------

@Configuration -> класа ще служи като factory за други Beans

public class ZooConfig{

//        @Primary -> за да покажем кой бий е с по-голям приоритет.

@Bean("cat")

public Animal cat(){

return new Cat();

}

@Bean("normalDog")

public Animal dog(){

return new Dog();

}

@Bean("mySuperDog") -> всеки един Bean си има като ппц взема името на метода.Но може да се зададе ръчно.

public Animal superDog(){

return new Dog(true);

}

//инициализацията на бийновете е Eager, дори и да не се извиква и използва, той ще бъде инициализиран.

@PostConstruct -> при инициализацията на всеки от бийновете се изпълнява този конструкт.

public void afterInit(){

Sout ("Dog is ready to bite!")

}

}

---------------

//Всички бийнове се инициализират от спринг при стартирането на приложението

//и започват да живеят в Application Context

@Service

public class ZooService{

private final Animal animal;

@Autowired

public ZooService(Animal animal){

this.animal = animal;

}

public void doWork(){

animal.makeNoise();

}

\*\*\*\*\* Ако работим с няколко Animal бийна\*\*\*\*\*

!!По този начин ще инжектира и използва всички необходими за работата бийнове в дадения лист.!!

//Не се гледа кой е праймъри а се инжектират всички.

private List<Animal> animals;

@Autowired

public ZooService(List<Animal>animals){

this.animals = animals;

}

public void doWork(){

animals.stream.forEach(Animal::makeNoise)

}

\*\*\*\*\* Работа с няколко бийна без колекция- SuperDog\*\*\*\*\*

Когато пак няма @Primary бийн, а искаме да ги използваме всичките

private final Animal animal1;

private final Animal animal2;

@Autowired

public ZooService(@Qualifier("mySuperDog")Animal animal1,

@Qualifier("normalDog") Animal animal2){

this.animal1 = animal1;

this.animal2 = animal2;

}

public void doWork(){

animal1.makeNoise();

animal2.makeNoise();

}

}

-----

@Component

public clas DemoZoo implements CommandLineRunner{

private ZooService zooService;

@Autowired

public DemoZoo(ZooService zooService){

this.zooService = zooService;

}

@Override

public void run(String...args) throws Exception{

zooService.doWork();

}

}

Тъй като всички бийнове живеят в ап контекста, ние дори можем да извлечем даден клас от него.

Пример:

@SpringBootApplication

public class mainApplication{

psvm(){

ApplicationContext context = SpringApplication.run(MainApplicaion.class, args);

Animal dog = context.getBean(Dog.class);

Sout("DOG:" + dog.getClass().getSimpleName());

}

}

3. Bean Scopes in Spring FW:

a/ Singleton -> by default -> в АП контекста има само една инстанция на даден бийн създадена от спринг

и навсякъде където го инжектираме използваме един и същ инстанс на дедния клас.

b/ Prototype -> еквивалента на New оператора, всеки път когато инжектираме някакъв клас, създаваме нова

инстанция.

c/ Request -> в рамките на 1 https request ще се инжектира винаги един и същи бийн.

d/ Session -> когато правим Web приложение, използваме Session Bean за да пазим неща които са

специфични за сесията на даден User.

@Bean

@Scope("singleton") -> може да пропуснем скоупа тук, тъй като използваме default-ния.

@Scope("prototype")

public Student student(){

return new Student;

}

Хубавото на Singleton и повечето бийнове е че те са stateless, което избягва доста проблеми понякога.

Prototype -> statefull

\*\*Bean Lifecycle\*\*

Instantion -> set Properties -> set Name -> set Application Context

->Post initialization ...destroying

Как можем да се захванем към тея life-cycle евенти на бийновете е:

@Component

private class DbInit{

private final UserRepository userRepository;

public DbUnit(UserRepository userRepository){

this.userRepository = userRepository;

}

@PostConstruct -> изпълнява се еднократно от Spring след инициализацията на даден бийн.

private void postConstruct(){

User admin = new User("admin", "admin password");

User normalUser = new User("user", "user password");

userRepository.save(admin, normalUser);

}

}

----------------

@Component

public class UserRepository{

private DbConnection dbConnection;

@PreDestroy -> изпълнява се веднъж преди Спринг да премахне нашия бийн от Application context-a.

public void preDestroy(){

dbConnection.close;

}

}

//Ако искаме да разберем какво е името на дадения бийн.

class A implements BeanNameAware{

@Override

public void setBeanName(String name){

Sout("The name of this Bean is " + name);

}

}

//Когато искаме преди затриването на бийна нещо да се случи.

class B implements DisposableBean{

@Override

public void destroy() throws Exception{

Sout("The bean will be destroyed..");

}

}

IOC container vs Application context:

IOC -> това което се казва на цялото приложение,

това което управлява Dependency Injection-a в Спринг.

App context -> конкретната структура от класове,

вътре в който се съдържат бийновете(мястото където живеят бийновете).

\*\*\*\*\*\*\*\*\*\*\*\*\* Spring Data \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--------------------------------------------------|

DataBase <-> Repository -> Service -> Controller |

^                                ^                         |

V V |>> View

Entities Models/DTO |

--------------------------------------------------|

1.Entity - еквивалентен клас на таблица в базите данни.

2.Repository -> интерфейси, служат за дърпане на инфо от базата данни.

Service -> Бизнес логиката се изпълняват в сървисите,

те приемат и връщат Модели/ така наречените ДТО-та.

Те ще бъдат получавани като входни и изходни данни към и от сървиса.

3.Controller -> Входните точки в нашите приложения.

Класовете, които обработват HTTP request-ите GET/POST/PUT/DELETE и тн.

Когато заявката дойде по HTTP, той ще вика Service,

а Service вътрешно в себе си ще работи с Repository-тата.

Repository-тата ще знаят как да се обръщат към базата.

Крайния резултат е така нареченото View, HTML страница,

която ще бъде рендерирана в браузъра.

или евентуално някакъв JSON файл, който ще се връща като респонс на REST API.

---------------------------------

в Gradle dependencies:

implementation 'org.springframework.boot:spring-boot-starter-data-jpa'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SPRING FUNDAMENTALS MVC \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Controller -> така се анотират бийновете контролери;

Слага се над определен клас, като обикновена практика е да слагаме контролер специфичен за даден

обект с всички необходими методи за него. Като контролера вика Сървисите.

@RequestMapping("/home")

public String home(Model model){

model.addAttribute("message","Welcome");

return "home-view"

}

@RequestMapping("/home")

public ModelAndView home(ModelAndView mav){

mav.addObject("message","Welcome");

mav.setViewName("home-view");

return mav;

}

- и двата метода правят 1 и също;

Проблема идва че не се специфицира какъв вид заявка ще очакват методите,

следователно ще отговаря на всички типове заявки.

Затова за да специфицираме какъв точно тип заявки се очакват, ние трябва да го опишем

@RequestMapping(value="/home", method=RequestMethod.GET)

Тъй като примера отгоре изисква малко повече писане и излишно повтаряне на методи и тн.

Спринг са измислили друг вид анотация която да специфицира:

@GetMapping("/home")

public String getHomeCatPage(){

return "cat-page.html";

}

@GetMapping("/dog")

@Respondbody -> автоматично кара Spring да сериализира обекта в JSON.

//Като не е задължително, защото връщането на обект директно се обръща в JSON в браузъра

//но за тази цел трябва да използваме @RestController на самия клас.

public Dog getDogHomePage(){

Dog dog = dogService.getBestDog();

return dog;

}

@PostMapping("/register")

public String register(UserDTO userDTO){

...

}

Можем да анотираме самия Контролер клас със @RequestMapping("/home")

След което методите @PostMapping / GetMapping и тн

няма нужда да съдържат пътя на URL-а в себе се.

Прави се с цел да намали писането и да е по-чисто

@Controller

@RequestMapping("/home")

public class HomeController{

@PostMapping("")

public ... method(DTO){

...

return "redirect:/home" -> след изпълнението да редиректнем към друга страница.

}

....

@GetMapping("")

...

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@GetMapping("/")

public String welcome(Model model){

model.addAttribute("name", "Pesho");

return "index";

}

-> Модел обекта ще бъде автоматично предаден към View като променлива.

задаваме името на view-то в ретърн-а.

Можем да използваме ModelMap modelMap вместо Model, като разликата е че

мапа е инстанция на Java LinkedHashMap в който можем да блъскаме стойности.

Можем да пасваме и връщаме ModelAndView както примера по-горе с него,

само че тук трябва да добавим име на модела и самото View,

като накрая връщаме самия ModelAndView обект.

@RequestParam annotation enables spring to extract input data that may be passed as a query, form data, or any arbitrary custom data.

параметъра се взема от querry String-a в url-a.

Този Стринг представлява обикновено "...?userId=1" след въпросителната.

В случая параметъра е 1.

При GetMapping-a вземаме тези параметри като дадем на @RequestParam-a някакво име ("id")

и зададем каква ще е променливата -> Long id) при което спринг взема key-value pairs и

ги инжектира в метода като аргументи.

Като може да се използва и при @PostParam/@PostMapping, но по-добрата практика е да изпращаме

директно ДТО-та.

Тези @RequestParam може понякога да ги няма, за да няма грешка в URL-a ние можем да

им променим default-ната стойност.

В примера ще сложим default-на стойност Annonymous. Ако няма зададен параметър, ще се използва тя:

..(@RequestParam(name = "author", defaultValue = "Annonymous")String author)

Можем също така да направим параметъра не задължителен/опционален/

@GetMapping("search")

public String search(@RequestParam(name="sort", required = false) String sort){}

Много често в практиката, параметрите не се съдържат в query String-a на URL-a

а се намират в пътя на URL-a:

localhost:8080/details/24

@GetMapping("details/{id}")

public String details(@ParamVariable("id") Long id){}

Когато използваме @Post методите ппц няма никаква валидация и е необходимо тя да бъде сложена допълнително.

ако искаме да редиректнем след някаква операция

можем да върнем:

return "redirect:/login";

като /login е някакъв друг енд пойнт.

@PostMapping("/register")

public String register(UserDTO userDto, RedirectAttributes redirectAttributes){

redirectAttributes.addAttribute("errirId", 3);

return "redirect:/login";

}

Spring ни предоставя на готово класа RedirectAttributes

ще ни върне към localhost:8080/login/errorId=3

error with id 3 means we are successfully logged in the app.

Ако искаме да задържим обектите след редирект:

public String register(@ModelAttribute UserDTO userDto, RedirectAttributes redirectAttributes){

...

redirectAttributes.addFlashAttribute("userDto", userDto);

return "redirect:/register";

}

тъй като след @Post заявката се използва 2ра, която е @Get, ако не искаме

данните за нея да бъдат изгубени трябва да го направим по горния начин за да

запазим обект инфото. Като използваме Flash атрибутите, като

реално се използва сесията на потребителя.

----------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\* State Management, Cookies and Sessions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.Кукитата са малки текстови файлове.

Те се раждат на сървъра и сървъра го изпраща към браузъра.

Като кукитата се изпращат в хедърите на http response-a.

Те държат малко парче данни за специфичен клиент.

За какво можем да ги използваме?

Основната идея е:

1.Session Management - login/ shopping card / game scores или всичко което

сървъра ще трябва да помни.

- Проблема е че HTTP протокола е stateless, което ще рече че всеки следващ рекуест не пази

никаква информация за предишните рекуести(не се предава state)

2.Персонализация - запазване опции за дадени сайтове ( смяна на тема и тн.)

3.Проследяване - записване и анализиране поведението на клиентите.

- С тях знаем дали потребителя е логнат или не, в кой акаунт е влязал,

да знаем примерно browser дейностите на сайта от потребителя, да знаем

и пазим информация за полета,които са попълнени преди това при регистрация и тн.

примерно правим заявка

GET / index HTTP/1.1

Cookie: lang=en

Set-Cookie header:

HTTP/1.1 200 OK

Set-Cookie:lang=en;

Когато браузъра види хедъра: Set-Cookie, хваща го и

създава едно куки от подадения key=value pair,

следващия път когато поискаме http индекс със стойността 1.1

браузъра автоматично ще прати кукито на сървъра при първия рекуест.

Като по този начин сървъра взема кукито и му дава дадените заложени стойности.

Get/index HTTP/1.1

S -----------------------> W

E        HTTP/1.1 200 OK          E

R        Set-Cookie: lang=en         B

V <----------------------- A

E        GET/index HTTP/1.1 P

R -----------------------> I

Cookie:lang=en

Кукитата имат специфична структура. Тя трябва да отговаря на някакви изисквания.

Не може да е свободен текст.

То се състои от Име, стойност, като има и атрибути които са опционални

те са от сорта на Domain=foo.com; Path=/test/;Expires=(кога кукито ще бъде изтрито автоматично)

... и тн.

Тези атрибути са key=value pair.

Kогато кукито отива обратно към сървъра, браузъра не праща тези атрибути,

те служат за допълнителни неща, които определят държанието на браузъра,

обратно към сървъра ще се върне SSID=... сесия ИД-то.

Домейна определя уеб сайта към който принадлежи това куки.

Expires - дефинира кога браузъра да изтрие даденото куки.

Max-Age - интервала от секунди преди кукито да бъде изтрито.

Security - Когато едно куки дадено то сървъра е секюрнато,

то трябва да бъде обратно върнато на сървъра само ако се използва

секюрната/криптирана конекция.

Ако браузваме в нормален сайт без да е секюрнат, то кукито няма да бъде

върнато обратно на сървъра.(session cookie, то е хубаво да се използва по такива

енкриптнати кънекции)

httpOnly дефинира че кукито не може да бъде пипано от JS.

\*\*Sessions\*\*

Какво са сесиите? - начин да запазим информация за потребител, която да бъде използвана

на няколко страници.

Web APP -> Session(user:Teo) ->/login; /home;...etc.

За да запазим тази информация, когато се логнем с username/password сървъра създава уникален номер,

който се нарича sessionId, който се изпраща в куки на клиента.

когато започнем да браузваме в сайта, това sessionId се връща от браузъра към сървъра.

сървъра трябва да има SessionStorage в който има Идта на user-ите и sessionId-тата

на всеки един request в хедърите към сървъра и така сървъра знае кой рекуест от кой акаунт идва.

\*\*\*\*\*\*\*\*\*\*\*\*\*\* REST API Spring \*\*\*\*\*\*\*\*\*\*\*\*\*

{

"name":"ivaylo",

"age":29

"sex":"MALE"

}

PUT - пълен ъпдейт: -> ivaylo, 29, NULL

{

"name":"ivaylo",

"age":29

}

PATCH - >частичен UPDATE -> ivaylo, 30, MALE

{

"age":30

}

@DeleteMapping("{id}")

public ResponseEntity<BookDTO> deleteBookById(@PathVariable("id")Long id){

this.bookService.deleteById(id);

return ResponseEntity.noContent().build();

}

public void deleteById(Long id){

bookRepository.deleteById(id);

}

---------

@PostMapping()

public<ResponseEntity<BookDTO> createBook(

@RequestBody BookDTO newBook

UriComponentsBuilder uriComponentsBuilder

){

long newBookId = bookService.createBook(newBook);

return ResponseEntity.created(uriComponentsBuilder

.path("/api/books/{id}").build(newBookId)).build;

}

public long createBook(BookDTO book){

Optional<AuthorEntity> author =

this.authorRepository.findAuthorByName(book.getAuthor().getName());

BookEntity newBook =

new BookEntity()

.setTitle(book.getTitle())

.setAuthor(author.isPresent() ?

author.get() : createNewAuthor(book.getAuthor().getName()));

bookRepository.save(newBook).getId;

}

private AuthorEntity createNewAuthor(String name){

return authorRepository.save(new AuthorEntity().setName(name));

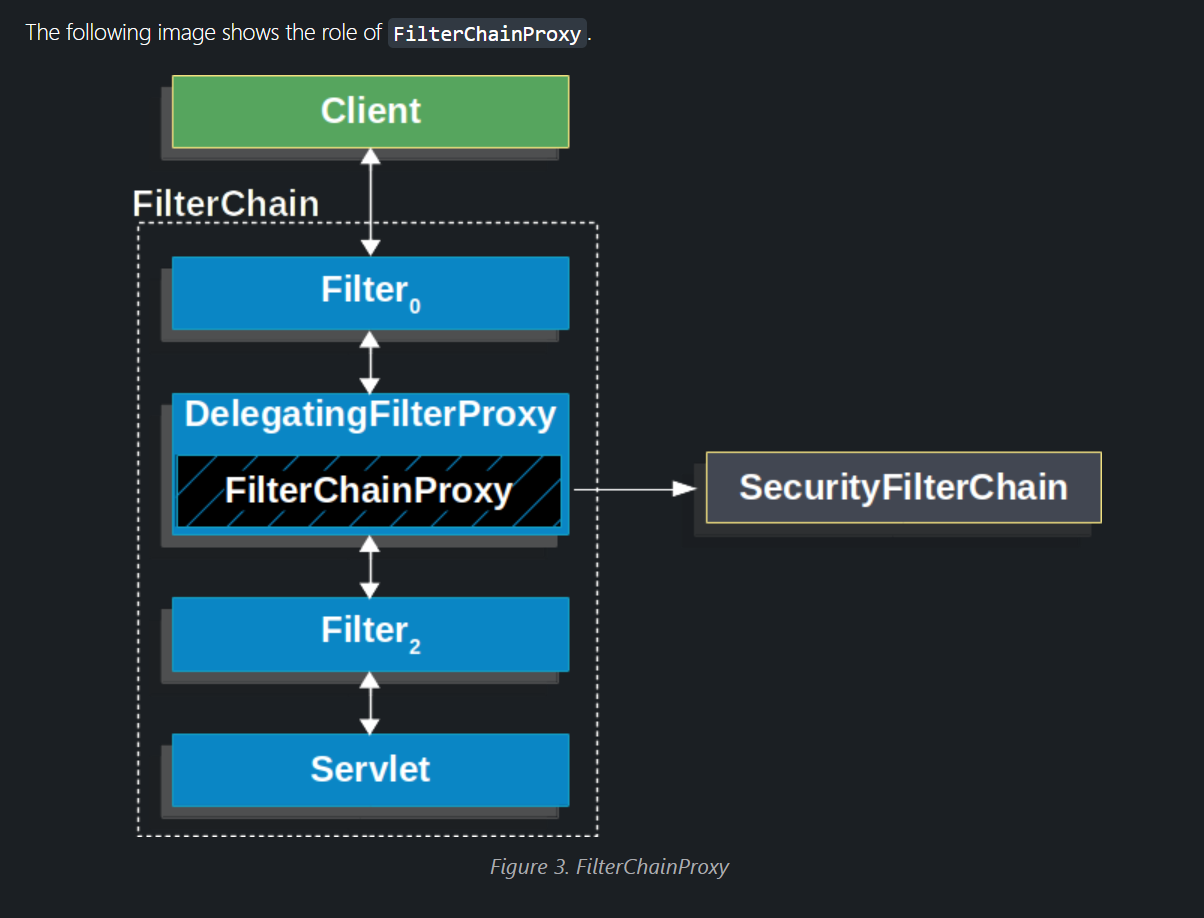
}

--------------

FilterChainProxy -> съдържа Spring Security’s Servlet support, той е специален филтър

предоставен от Спринг секюрити който позволява делегирането на много инстанции на филтри през SecurityFilterChain. Тъй като FilterChainProxy е бийн е типично той да бъде

обвит(wrapped) в DelegatingFilterProxy.



DelegatingFilterProxy -> спринг предоставя имплиментация на филтър наречена DelegatingFilterProxy,

която позволява свързването между Servlet контейнер и ApplicationContext-a на Spring.

Servlet контейнерът позволява регистрирането на Filter инстанции, използвайки собствени стандарти, но не знае за Спринг дефинираните бийнове.

FilterChain-а съдържа всички дефинирани филтри бийнове от спринг + дефинираните от нас филтри които се намират в DelegatingFilterProxy.

\*\*\*\*\*\*

Имаме дефинирани филтри от спринг, които са бийнове, те се намират в FilterChain-а,

отделно имаме DelegatingFilterProxy което можем да го вмъкнем в FilterChain-а,

което съдържа в себе си SecurityFilterChain, който от своя страна е списък от

секюрнати филтри които са дефинирани от нас.

Секюрнатите филтри в SecurityFilterChain-а са типично бийнове, но те се регистрират с FilterChainProxy, а не с DelegatingFilterProxy.

Предимствата на FilterChainProxy са че предоставя стартова точка на всичкия Спринг секюрити Сървлет съпорт.

Поради тази причина ако се опитате да трабълшутнете FilterChainProxy е чудесна точка да започнете.

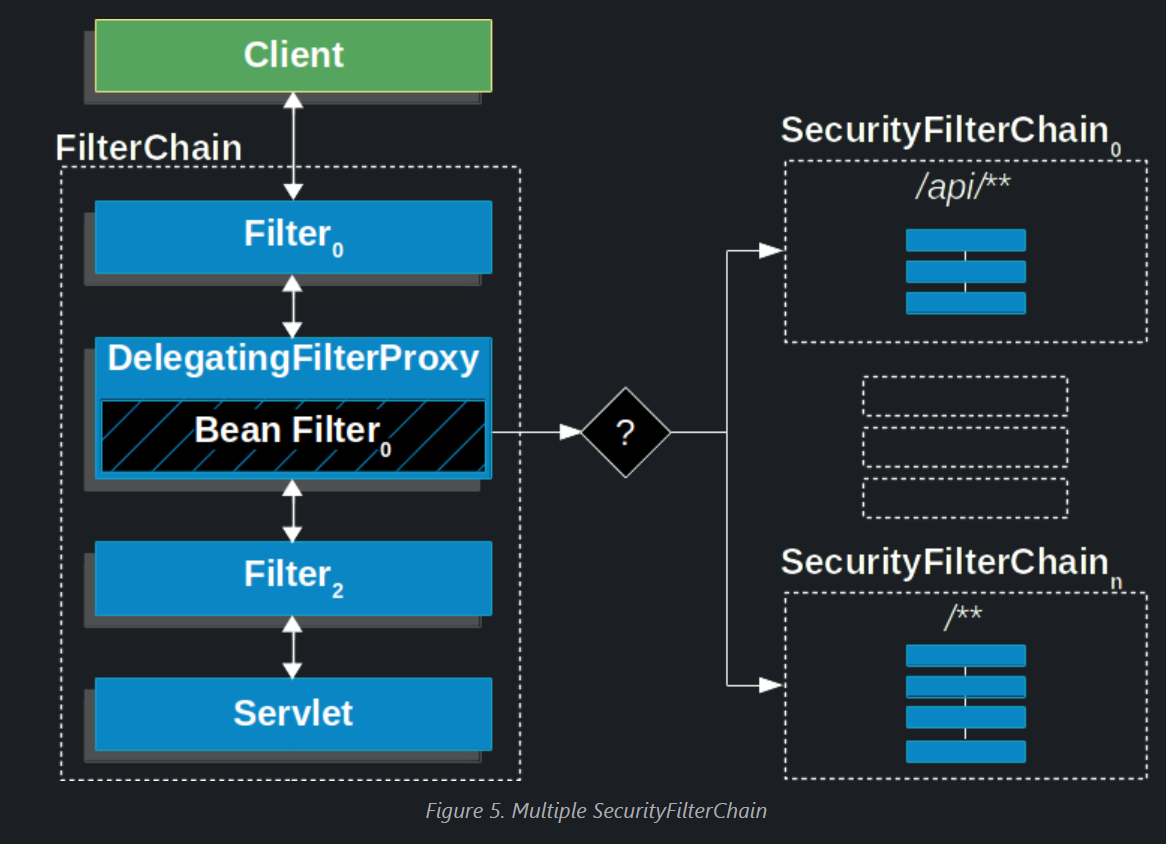
Тъй като FilterChainProxy е централен за използването на Spring Security може да изпълнява задачи които не са опционални. Например изчиства SecurityContext-а за да избегне memory leaks.

Също използва Spring Security’s HttpFirewall за да защити апикациите срещу атаки.

В допълнение предоставя гъвкавост при определянето кога SecurityFilterChain трябва да бъде извикан.

В Servlet контейнер, филтър инстанциите се извикват базирано на URL-a, обаче FilterChainProxy може

да определи извикване базирано на HttpServletRequest използвайки RequestMatcher интерфейс.



При няколко SecurityFilterChain-а, FilterChainProxy решава кой SecurityFilterChain трябва да бъде използван. Първият SecurityFilterChain който пасва ще бъде извикан.

Ако URL-а на /api/messages/ е поискан, това ще пасне с първият елемент с индекс 0 -> SecurityFilterChain(0) с патърн /api/\*\* , ако никоя от инстанциите на

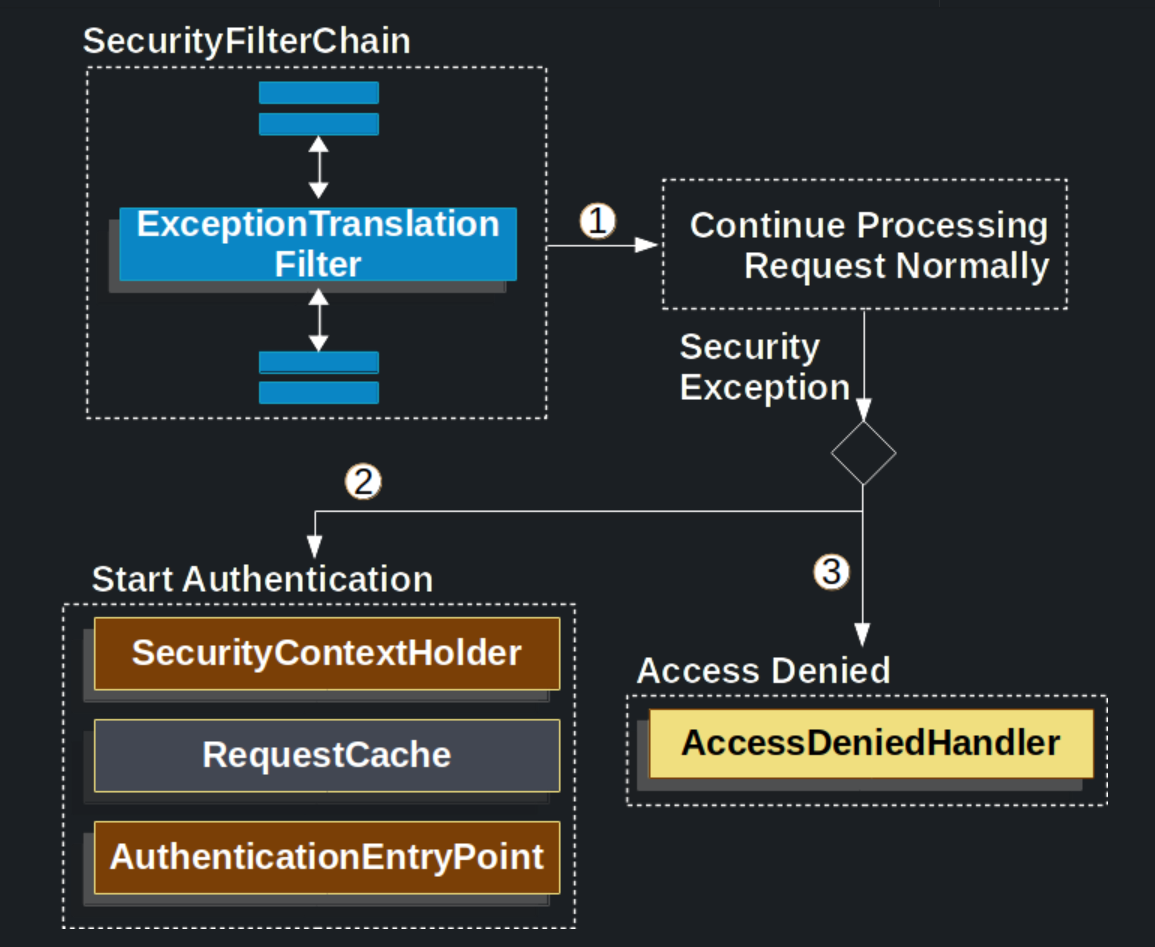
SecurityFilterChain не пасне на рекуеста, последната инстанция ще бъде извикана SecurityFilterChain(n).

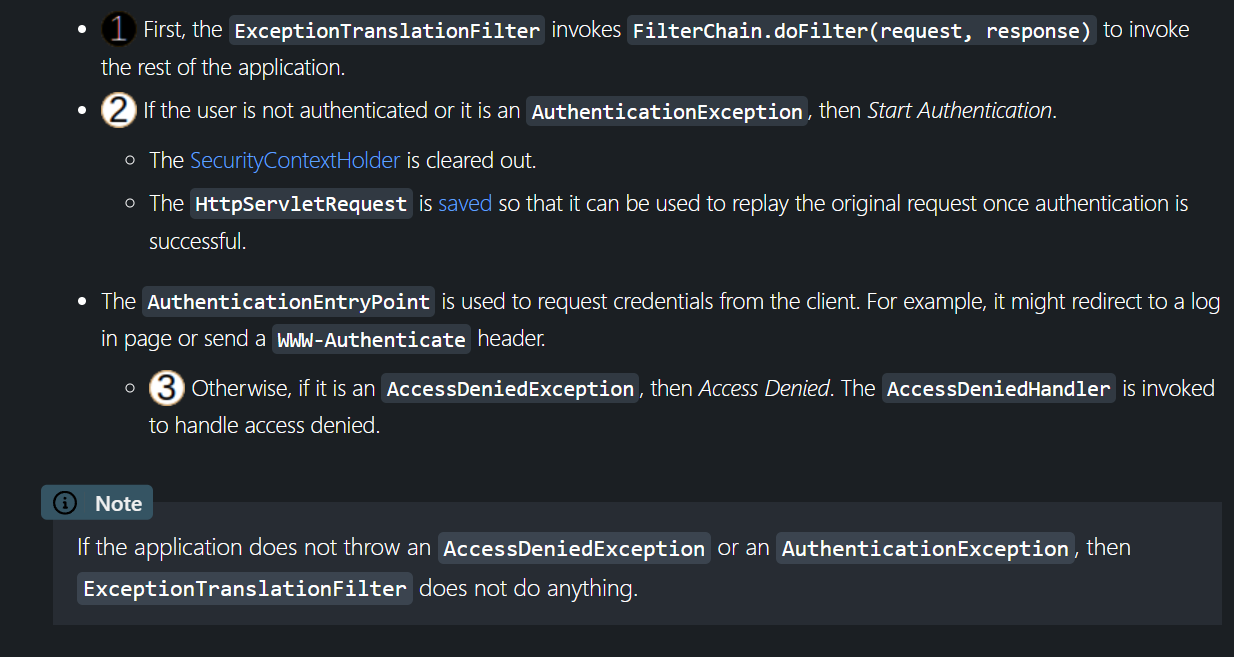
Важно е да се отбележи че всеки SecurityFilterChain е конфигуриран индивидуално и уникално, като е изолиран от другите, може да съдържа различен брой филтри.

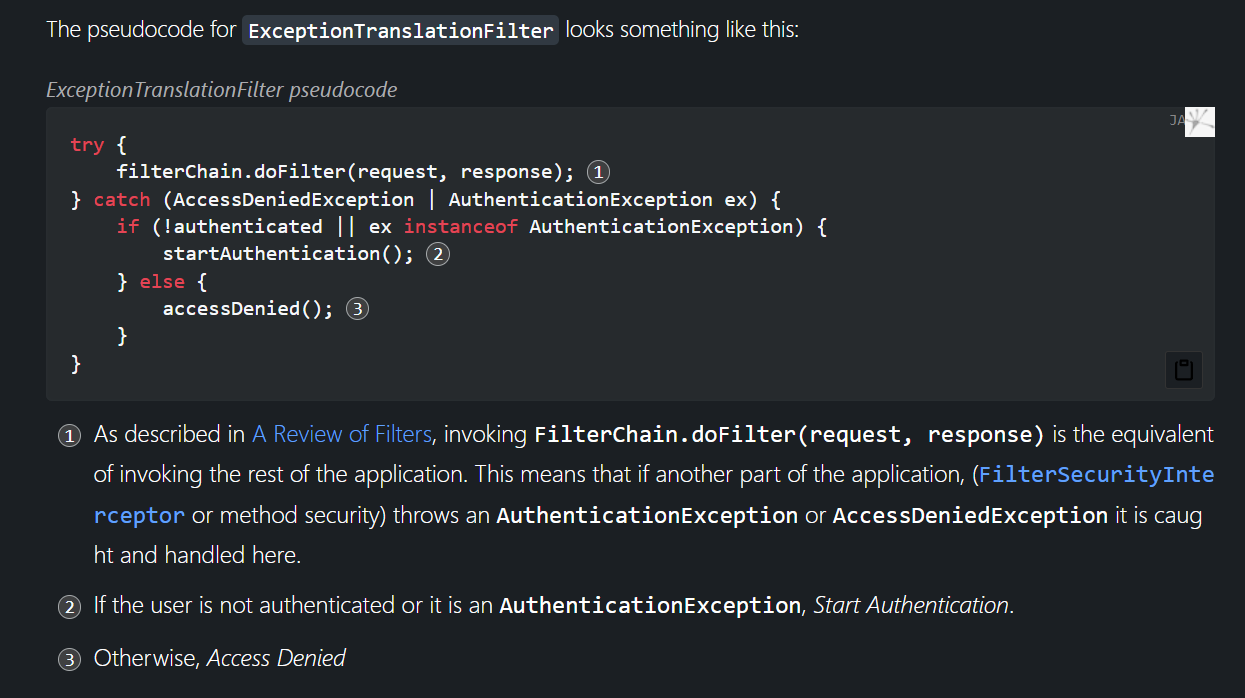
В случай че искаме Спринг секюрити да игнорира даден рекуест, можем да създадем SecurityFilterChain, който в себе си няма да има никакви филтри.

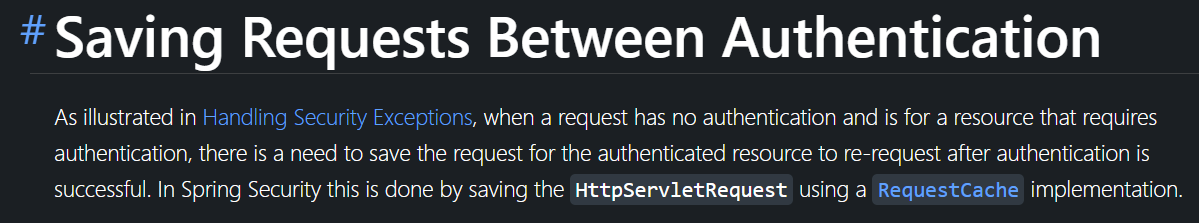
Security Filters са добавени в FilterChainProxy с SecurityFilterChain API. Подредбата на филтрите има значение. Типично не е необходимо да се знае подредбата на Spring Security филтрите. Обаче има моменти в които е от полза да се знае подредбата.

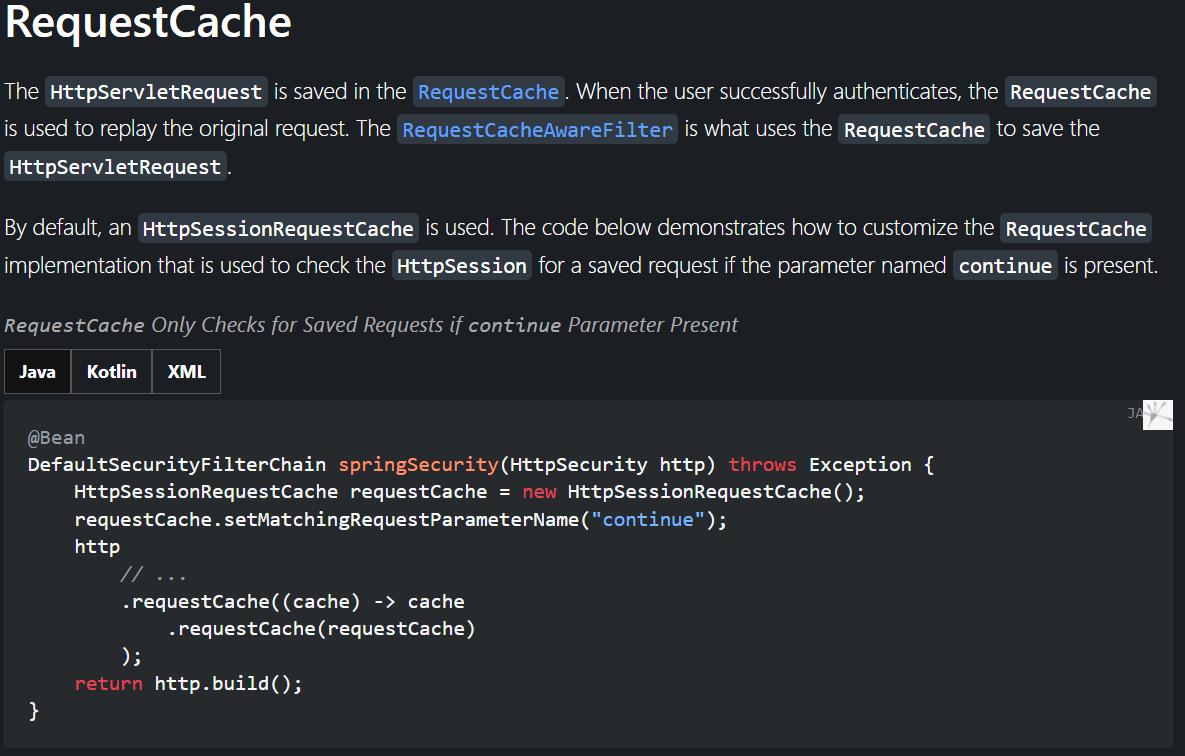
<https://docs.spring.io/spring-security/reference/servlet/architecture.html>

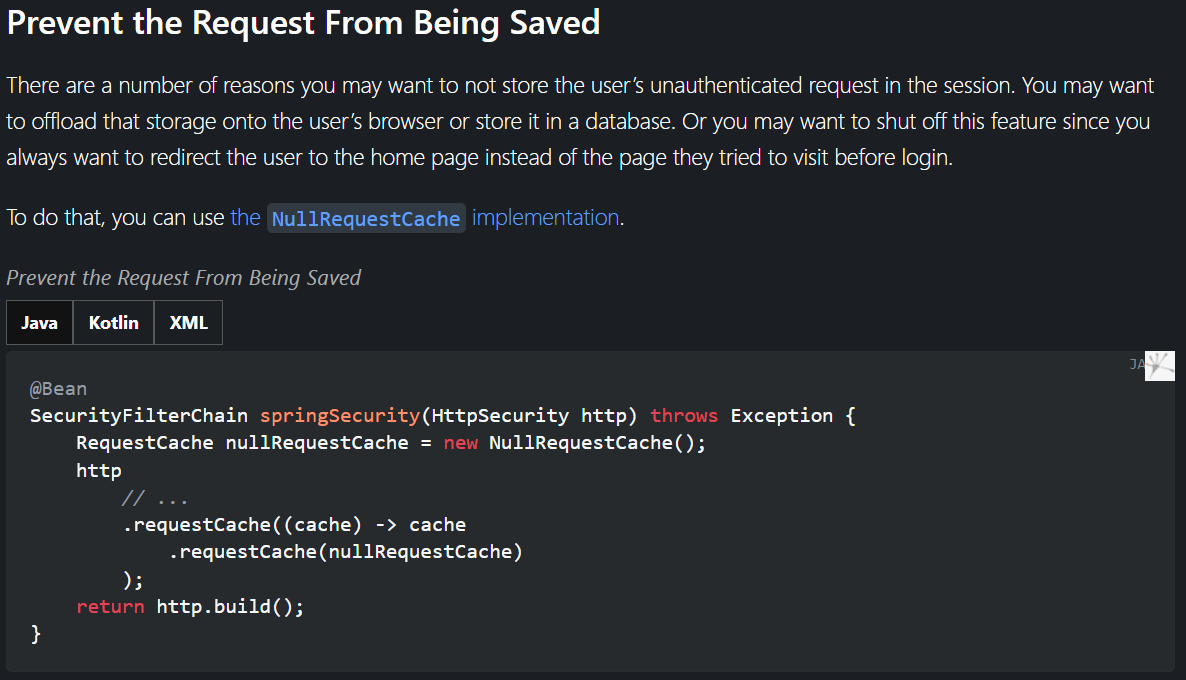


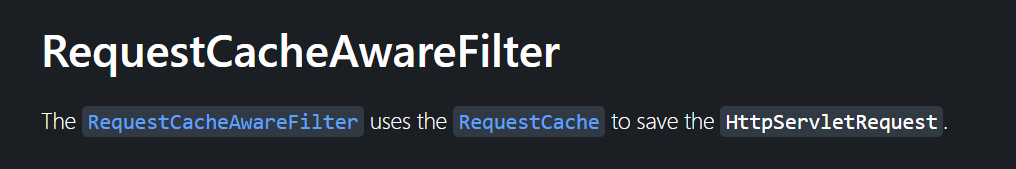












\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*SERVLET AUTHERNTICATION ARCHITECTURE\*\*\*\*\*\*\*\*\*\*\*

SecurityContextHolder -> тук спринг секюрити държи детайлите на всички идентифицирани.

SecurityContext -> се ползва от SecurityContextHolder и съдържа Authentication на текущия удостоверен потребител.

Authentication -> може да бъде вход към AuthenticationManager за да предостави credentials които user-a е предоставил за да се удостоверят или текущия user от SecurityContext-а.

GrantedAuthority -> правомощия които се предоставят на principal-a при удостоверяването(roles, scopes etc.)

AuthenticationManager -> API-то което дефинира как Spring Security филтрите изпълняват удостоверяването (Authentication) ;

ProviderManager -> най-често срещаната имплиментация на AuthenticationManager;

AuthenticationProvider -> използван от ProviderManager да изпълни специфичен вид authentication/удостоверяване.

Request Credentials with AuthenticationEntryPoint -> използва се за поискване на credentials от клиента,

(Редиректва към login page или изпраща WWW-Authenticate response. Etc)