

1. Четене/закръгляне на числа/видове променливи/прости операции/кастване/цикли

Четене от конзолата:

- Чрез Scanner

```
Scanner sc = new Scanner(System.in);
String input= sc.nextLine();

Integer.parseInt(sc.nextLine());
Double.parseDouble(sc.nextLine()); - и т.н. за byte, long...
```

sc.nextInt() – чете на същ или нов ред, ако имаме вход от цели числа от повече от 1 елемент

- Чрез BufferedReader – за по-голяма бързина

```
public class ReverseArray {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(reader.readLine());
        String[] elements = reader.readLine().split("\\s+");
    }
}
```

Шаблони

%n – на нов ред навсякъде – използваме в шаблон

\r\n или \r или \n в различните операционни системи – ТОВА РАБОТИ БЕЗ ШАБЛОН !!!

“\r\n” - Windows

“\n” - Unix

System.lineSeparator() – ТОВА РАБОТИ БЕЗ ШАБЛОН !!! за всички операционни системи

```
return String.format("%s %s - %s%n%s", this.getFirstName(), this.getLastName(),
    this.getJobTitle(),
    this.getProjects()
        .stream()
        .map(p -> p.toString())
        .sorted((f, s) -> f.compareTo(s))
        .collect(Collectors.joining("\n\t"))); //ТОВА РАБОТИ БЕЗ ШАБЛОН !!! за всички
```

операционни системи

System.out.printf("%.3f%n", result); три знака след десетичната запетая, дясно подравнено с 3 цифри след десетичната запетая

```
System.out.println(String.format("%.3f%n", result))
```

System.out.printf("%.0f", result); изписва без десетичната запетая

System.out.printf("%d:%02d", finalHour, finalMinutes); - дясно подравнено с две цифри винаги(Ако няма някои или всички от цифрите, то слага 0 за всяка цифра) - изписва с 1 нула пред числото ако то е едноцифрен

```
String format = String.format("%02d:%02d", minutes, secs);
```

System.out.printf("%.2f%%%n", p1Percents); - %n e new line + процент да ни се изписва + един за escape

```
String name = String.format("%.3f%n", result);
```

String text = new DecimalFormat("0.#####").format(5,33437) – нулата определя нула/число със сигурност, # определя че може да има цифра до 4 знака ако числото е толкова дробно или по-малко от 4 при знака(да си трае), do not display trailing zeros

```
DecimalFormat decimalFormat = new DecimalFormat("0.#");
System.out.print(decimalFormat.format(5.0)) -> връща 5
```

In yellow – to check once more

2ри вариант за премахване на нули (salary е от тип double/Double):

```
String.format("%s %s gets %s leva", this.firstName, this.lastName, this.salary);
```

Padding отпред:

На числа - Padding left with zeros: - System.out.printf("|%03d|", 93); // prints: |093| - дясно подравнено с 3 цифри винаги. Ако няма някои или всички от цифрите, то слага 0 за всяка цифра

На текст - String of specified length (involves max and min) - You want left justified so "-N" instead of N for first value

```
System.out.printf("|%-15.15s|", "Hello World"); //|Hello World|
```

"%,d" - thousand separator или 1,000

System.out.printf("%s", text); е същото като System.out.println(String.format("%s", text));

Като може да използваме и конкатенация в самия String.format:

```
System.out.println(String.format("Source: %s%n" + "Destination: %s%n" + "Spare: %s%n",
                                 stringA, stringB, stringC));
```

Placeholders

%s или %S (String); - с малки или големи букви

%d (int);

%f (double);

%c (char) или %C; - с малък или голям Char

%b или %B (boolean) ...

%n – new line

Основни числови данни

Default стойности за реално число:

Is 0.0F for the float type

Is 0.0D for the double type – по подразбиране реално число е double

Floating-point types are:

- float ($\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$) - 32-bits, precision of 7 digits
- double ($\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$) - 64-bits, precision of 15-16 digits

Default стойности за цяло число:

long number = 1L;

Type	Default Value	Min Value	Max Value	Size
byte	0	-128 (-2 ⁷)	127 (2 ⁷ -1)	8 bit
short	0	-32768 (-2 ¹⁵)	32767 (2 ¹⁵ - 1)	16 bit
int	0	-2147483648 (-2 ³¹)	2147483647 (2 ³¹ - 1)	32 bit
long	0	-9223372036854775808 (-2 ⁶³)	9223372036854775807 (2 ⁶³ -1)	64 bit

2.35e+24 числото на 2,35 на 10 на 24та степен

2.35e-24 числото на 2,35 на 10 на 24та степен

'0x' and '0X' prefixes mean a hexadecimal

IEEE 754 - IEEE Standard for Floating-Point Arithmetic – има разлика след десетичната запетая/губим данни

Very high precision is BigDecimal

```
BigDecimal number = new BigDecimal(0);
BigDecimal num = new BigDecimal(sc.nextLine()); или строка
number = number.add(BigDecimal.valueOf(2.5));
number = number.subtract(BigDecimal.valueOf(1.5));
number = number.multiply(BigDecimal.valueOf(2));
number = number.divide(BigDecimal.valueOf(2));
```

//BigInteger закръгления стават и чрез този package

```
package java.math;  
o.getPrice().setScale(2, RoundingMode.HALF_UP));
```

Има също и BigInteger

```
BigInteger number = new BigInteger(String.valueOf(1));
BigInteger C = A.add(BigInteger.valueOf(val));
B = new BigInteger("123456789123456789");
if (a < b) {} // For primitive int
if (A.compareTo(B) < 0) {} // For BigInteger
```

```
int a = 1;  
double b = 2.4, c = 2.4;  
  
option 1) a = a + (int)(b);  
option 2) a+= b; // a = (int)(a + b);
```

```
char symbol = sc.nextLine().charAt(0);
String architectName = sc.nextLine();
int numberOfProjects = Integer.parseInt(sc.nextLine());
double percents = Double.parseDouble(scanner.nextLine());
```

7 / 3 - ако е Integer връща цяло число 2

```
Math.round(45.67852); //46 – хвърля int
```

```
Math.round(45.37852); //45 – хвърля int
```

Math.floor(45.67); //45.0 - хвърля double като изчиства 45,00000000000000000000003

```
Math.pow(2, i); // хвърля Double 2 на коя степен
```

Type conversion / кастване:

- от по-малък в по-голям тип данни - не губим данни (**Имплицитно**):

```
double a = 4;
```

- от по-голям/широк в по-малък/тесен тип данни

int a = (int)5.66; - връща 5 - експлицитно, и губим данни

Кастване от малък int в башин Integer

```
int currentValue = current.element;
this.current = this.current.prev;
return (Integer) java.lang.Integer.valueOf(currentValue);
```

ВАЖНО: когато сравняваме променливи от различен числов тип, то сравнението работи

```
double a = 6.4;
int b = 6;
if (a < b) {
    System.out.println(a < b);
} else {
    System.out.println(a < b);
}
```

Math.PI

```
int maxNumber = Integer.MIN_VALUE;
int nMin = Integer.MAX_VALUE;
String a = scanner.nextLine();
String b = scanner.nextLine();
System.out.println(a.equals(b)); //True
!a.equals(b); - отрицание
System.out.println(a.equalsIgnoreCase(b)); //True
boolean greaterAB = (a > b);
```

&& - конюнкция

|| – дизюнкция

^ - изключващо Или (XOR)

! – отрицание (!F = T)

!= различно

++a – първо променям стойността и след това използвам променливата на същия ред

a++ - първо се изпълнява на същия ред със старата стойност, и след това ще се смени стойността на a.

```
int a = 5, b = 5;
a += 2.5; //връща 7 като изрязва десетичната част без да пита
b = (int) (b + 2.5); //връща 7 като казва, че трябва да кастнем и да изрежем десетичната част
```

В цикъл:

Break; - прекъсва текущия цикъл

Continue; - праща цикъла в следващо завъртане, без да изпълнява останалото текущо тяло на цикъла

Унарни операции – 1 аргумент е нужен

Бинарни операции – два аргумента са нужни

Тернарен оператор – 3 аргумента са нужни - пример

```
int[] numArr1;
int[] numArr2;
int maxLength = (numArr1.length > numArr2.length) ? numArr1.length : numArr2.length;
```

Инструкция на процесора можем да кажем, че е код завършващ с точка и запетая накрая.

Реално повече инструкции на процесора има на 1 ред код -аритметични/логически и т.н.

Обект от бащин тип може да се съхранява null + малък тип

Integer -> int

Double -> double

Character -> char

```
while (!(product = sc.nextLine()).equals("End")) {
```

```
}
```

```
const = final
```

hardcore value – предварително зададени входни данни, а не данни зависещи от потребителя

```
String s = sc.nextLine();
switch (s){
    case "Az": break;
    case "Ti": break;
    default: break;
}
```

Switch expression – valid for sure from Java 17 SDK

```
String commandOutput = switch (commandName) { //може да присвоява резултат
    case REGISTER_USER_COMMAND -> {
        RegisterDTO registerData = new ReggisterDTO(commandParts);
        User user = userService.register(registerData);

        yield String.format("%s was registered", user.getFullName()); //все едно return
    }
    // case LOGIN_USER_COMMAND -> userService.login();
    // case LOGOUT_USER_COMMAND -> userService.logout();
    default -> "Unknown command";
};
```

2. Базова работа със String и Char

Character Data Type – държи символ Unicode или част от Unicode

'\0' – default value

Escaping Characters – използваме \ само при специални и системни символи

```
char symbol = 'a'; // An ordinary character
symbol = '\u006F'; // Unicode character code in a // hexadecimal format (letter 'o')
symbol = '\u8449'; // 葉 (Leaf in Traditional Chinese)
symbol = '\''; // Assigning the single quote character
symbol = '\\'; // Assigning the backslash character
symbol = '\n'; // Assigning new line character
symbol = '\t'; // Assigning TAB character
```

Стингове:

null – default value

String destination = ""; //празен стринг

Integer.parseInt("") + num.charAt(i)) чар плюс стринг прави стринг – същото като
String.valueOf('a')

```
String toRepeat = "abc";
String temp = temp.concat(toRepeat);
```

```
String text = scanner.nextLine(); // въвеждаме SoftUni
int length = text.length(); // 7 за string

String current = expression.substring(2, i+1);
```

```

От число към String
String newNumber = number + "";
String newNumber = Integer.toString(25); //връща 25 като стринг

Integer.parseInt("") + CharSymbol or number); - правим го на String
Character.getNumericValue(CharSymbol); - правим го на String

String currentNumber = "" + i;
String currentnumber = String.valueOf(i);

Махаме всички удивителни от стринга, и го разделяме на масив
String currentInput = sc.nextLine().replaceAll("!+", "");
currentArr = currentInput.split("");

System.out.println(Character.isDigit('3')); - връща true – проверка дали даден символ е число
boolean a = Character.isDigit(input[i].charAt(0));

"textbrat".toUpperCase();

if (a.compareTo(b) > 0) - а и б са Strings / низове

String output = input.charAt(input.length() / 2 - 1) + " " + input.charAt(input.length() / 2); - ако
първо събираме символите, то ще получим число, а не конкатенация на String

split(" ") – един Space

split("\s+") – повече Space/Tab/Enter/Return (white space) – пишем \\ за да обозначим само една наклонена \
Понякога, когато използваме \s+ се получава, че един от разделения елемент и е празен Space "". Него
премахваме с result.remove("");

sc.nextLine().split("\\s+", 2); - докато види първият WhiteSpace е първата част, останалата част е
втората част

.trim() – маха Space символите отпред и отзад на стринга – ако има такива Space-ве.

Whitespace – Space, Tab, Enter, Return

```

3. Бързи команди в средата IntelliJ

Alt + Enter – мулти тул в IntelliJ / средата за разработка IntelliJ предлага варианти / замества If със Switch / предлага създаване на метод по Дедукционния метод – от общото към частното / introduce local variable (или .var)

Ctrl + / - слага закоментар на избраните редове
`/* some text */` - закоментар на текст – вариант 2

Ctrl + Alt + L – форматира автоматично

Ctrl + Shift + V – последните наши копирания в IntelliJ

Ctrl + D – копира същото на нов ред

Shift + F6 - Refactor - сменя името на една променлива навсякъде в програмата

Debugger – с F8 минаваме на всеки ред, слагаме breakpoint (червена точка), с F9 отиваме до следващ breakpoint (червена точка)

Ctrl + (Alt) + B (Ctrl + click) – къде се използва този метод в кода ИЛИ Go To → Declaration and Usages

Alt + мишката + стрелки нагоре / надолу – пишем на няколко реда едновременно

Ctrl + Shift + стрелки нагоре надолу – места маркираното

Ctrl+C – копирам целият ред
Ctrl + X – Cut-ва целият ред
Ctrl + V – Поставя целият ред
Ctrl + P – при създаване на инстанция на класа чрез конструктор, показва параметри от какъв тип и в какъв ред трябва да има; дава инфо за get / set параметрите при списъци
Fori плюс Tab и ни задава цялото тяло на for цикъла
Alt+Enter и след това Iterate – създава FOREACH цикъл с тази колекция/масив/списък
Ctrl+Alt+M – Extract method – след селекция на даден код, да го сложим в метод този код
Ctrl+Shift+Enter – слага къдрави скоби където е пропуснато
Iter – Прави For each цикъл автоматично
Ctrl + Q – задава ни какво да въвеждаме в скобите

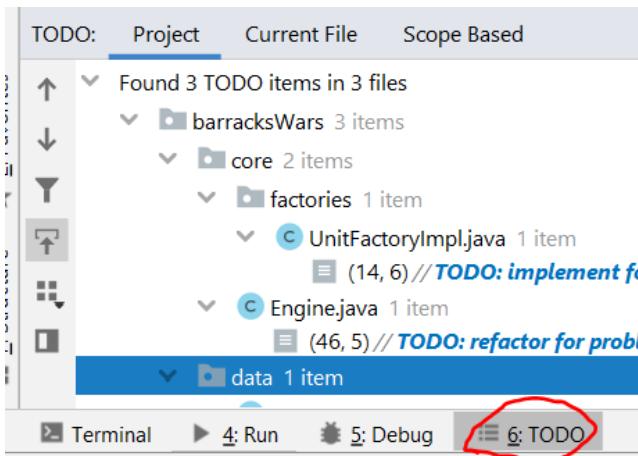
Ctrl + Q – вади документацията

Fn + F8 – при дебъгване
Ctrl + C – копира целият ред
Alt + задържан ляв бутон на мишката и движим нагоре/надолу - можем да пишем на няколко реда едно и също едновременно
Shift + scroll с мишката – хоризонтално движение
Ctrl + посочване без кликане върху променлива – показва информация за нея/типа ѝ, т.н.
Ctrl + клик върху променлива, преди да сме въвели израза – показва информация за нея/типа ѝ, т.н.
Като запиша .stream, и автоматично става Arrays.stream()
Като запиша .var, и автоматично създава променливата
Ctrl + Shift + V - показва последните копирания. Ако искаме да копираме предпоследното, да не се разхождаме постоянно
В режим Debugging – с посочване на курсора върху променлива, излиза все едно менюто debug отдолу което е.
Ctrl + Alt + O – премахва ненужни импорти от проекта
Ctrl + Shift + U – прави ги на малки или големи букви каквото си маркирал
Ctrl + Shift + N – при голям проект, за по-лесно търсене на файлове и класове
Alt + F7 – Find usages of a method/variable
Show UML Diagram – показва структурата от класове/обекти визуално
Ctrl + Alt + O – премахва не нужните импорти
Alt + J – маркираме име/текст, след това натискаме няколко пъти Alt + J и то ни селектира същия текст ако го има на друго място.
<https://www.diffchecker.com/> – от интернет, за сравняване на текст
Shift + Tab = Back Tab

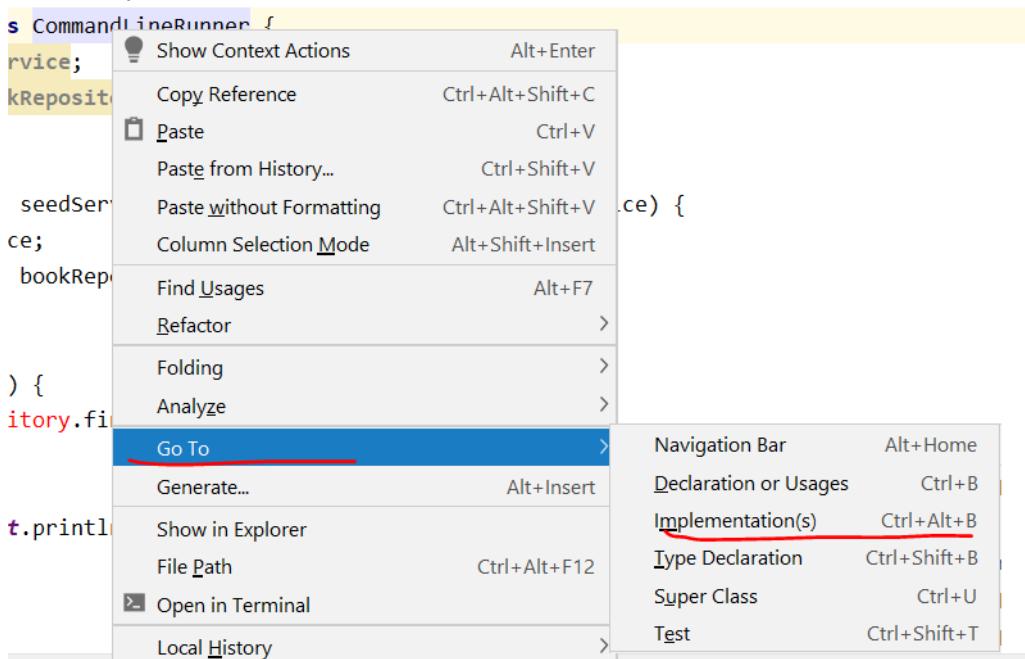
Ctrl + Alt + left – връща ни на предишното място на курсора от предходния файл

Snippets - Live Template – при маркиране на текст, за бърз достъп след това (от Tools):
Psvm – **public static void** main(String[] args)
Sout – System.out.println("January");
Sc – Scanner sc = new Scanner(System.in);

Може да си слагаме //TODO-та в кода, който пишем, и оттук имаме бърз достъп до тези редове



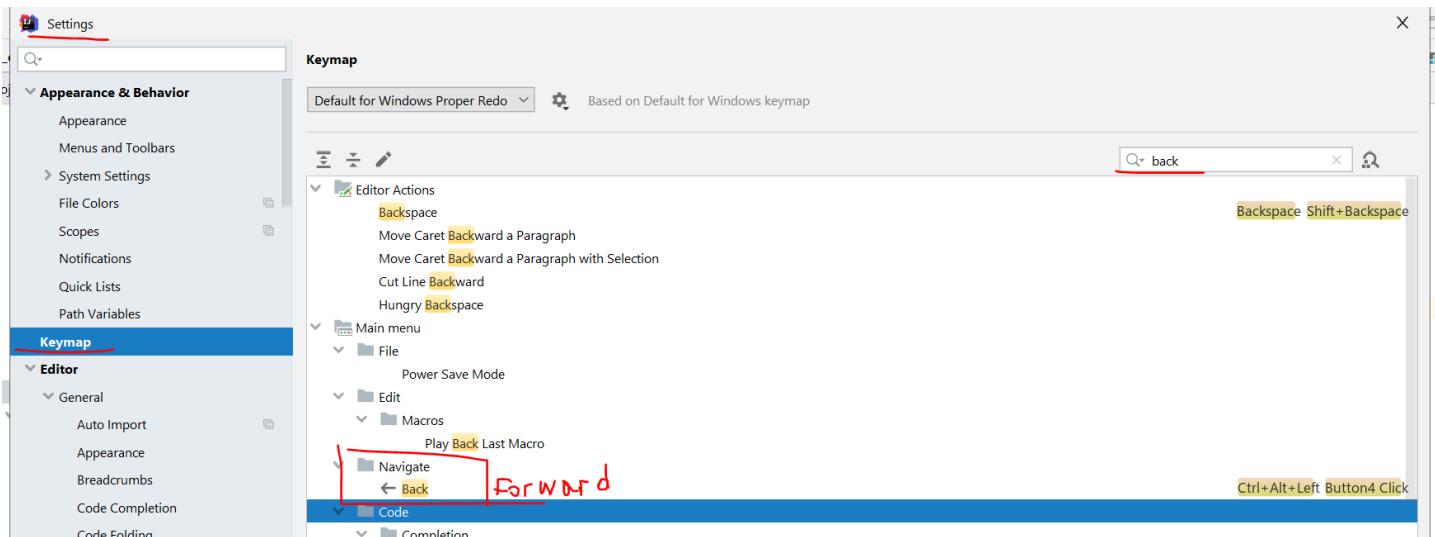
Go To Implementations



Ctrl + Tab задържаме – излизат ни всички отворени до момента табове

Ctrl + Alt + Left/Right (можем и сами да си нагласим клавишните комбинации)

Опция къде е бил последно курсора – наглася се от backward/forward – когато сме Ctrl+Click-нали да отидем в друг клас и искаме веднага да се върнем на предходния клас/курсора:



```
/** и enter - вмъкване на описание за даден метод
 */
*
* @param commandParts
*/
```

4. Масиви / Arrays

4.0. Работа с обекти в Java

```
var result = new Object[]{modifiedA, modifiedB};
Object[] objects = {"dqa", "cqad", "edq"};
new Object[size];
```

4.1. Четене на масив

`int[] numbers = new int[5];` - задаваме дължина 5 елемента на масива – **всеки елемент в момента е НУЛА!!! Това е default стойността на integer масива!**

```
int[] numbers = new int[] {1, 2, 3}; или int[] numbers = {1, 2, 3};
String[] day = new String[7];
String[] numberString = new String[] {"13", "42", "69"};
Или String[] numberString = {"13", "42", "69"};
```

`String[] numberStrings = sc.nextLine().split(" ")`; - Четене на ред със стрингове
`String[] parts = sc.nextLine().split("\s+");` - четене на ред от стрингове когато има повече Space разстояние между символите

Четене на масив / ред от числа:

```
int[] numbers = Arrays.stream(sc.nextLine().split(" ")).mapToInt(e -> Integer.parseInt(e)).toArray();
int[] numbers = Arrays.stream(sc.nextLine().split(" ")).mapToInt(Integer::parseInt).toArray();
```

Когато е от башин тип

```
Double[] acc = Arrays.stream(sc.nextLine().split("\s+"))
    .map(Double::parseDouble)
    .toArray(Double[]::new);
```

Четене на масив от числа по дългия начин:

```
String[] input = sc.nextLine().split(" ");
int[] numbers = new int[input.length];
for (int i = 0; i < numbers.length; i++) {
```

```

    numbers[i] = Integer.parseInt(input[i]);
}

char[] input = sc.nextLine().toCharArray();

Дължина на масив:
input.length;
Array.getLength(input); - по друг начин взима дължината на масива

```

4.2. Операции с масиви

Размяна на елементи

```

for (int i = 0; i < numbers.length / 2 ; i++) {
    int oppositeIndex = numbers.length - i -1;
    int oldNumbersI = numbers[i];
    numbers[i] = numbers[oppositeIndex];
    numbers[oppositeIndex] = oldNumbersI;
}

```

Копие на масив – опция 1 – създават се други/втори данни в heap-а/ в паметта за клонирания масив :

```

int[] arr = {1, 2, 5};
int[] copyArr = arr.clone(); // на ново място в паметта
copyArr[0] = 0;
copyArr[1] = 0;
copyArr[2] = 0;
System.out.println(Arrays.toString(arr));
System.out.println(Arrays.toString(copyArr));

```

Изменяме даден масив по дължина и стойност – Class Arrays или клас System

```

int[] num = new int[8];
int[] condensed = new int[5];
int[] num = Arrays.copyOf(condensed, condensed.length); - копираме масив с по-къса дължина от
началния – създава се нов масив, който сочи към различно място в паметта
num = condensed; - num е референция на същия масив condensed, който се намира на същото място в
паметта/heap-а 😊

```

```

arR = Arrays.copyOfRange(arR, index+1, length); - копираме масив след кой индекс и с
каква дължина и го присвояваме след това в същият

```

Можем да използваме и долната команда:

```
System.arraycopy(...);
```

Изменяме/увеличаване даден масив по дължина елегантно:

```
result = expandAndAddToArray(result, 5);
```

```

private static int[] expandAndAddToArray(int[] oldArray, int newElement) {
    int[] newArray = new int[oldArray.length + 1];
    for (int j = 0; j < oldArray.length; j++) {
        newArray[j] = oldArray[j];
    }
    newArray[newArray.length - 1] = newElement; // сложи на последната позиция
    return newArray;
}

```

Промяна подредба на елементите в масив

```

private static void exchange(int[] array, int index) {
    int[] temp = array.clone();
    int count = 0;
    for (int i = index + 1; i < temp.length; i++) {
        array[count] = temp[i];
        count++;
    }

    for (int i = 0; i <= index; i++) {
        array[count] = temp[i];
        count++;
    }
}

```

Конкатениране на 2 масива

```

String[] both = ArrayUtils.addAll(first, second);
ИЛИ
private static String[] exchangeArr(String[] arr, int index) {

    String[] a = Arrays.copyOf(arr, index+1);

    int length = arr.length;
    String[] b = Arrays.copyOfRange(arr, index+1, length);

    String[] both = Stream.concat(Arrays.stream(b), Arrays.stream(a))
        .toArray(String[]::new);

    return both;
}

```

Динамична памет:

- като дадем new - създаваме променлива, която достъпва масива
- като копираме масив в нова променлива copyNumbers - спирате да имаме достъп до елементите от стария масив

```

int[] numbers = new int[8];
int[] copyNumbers = {1, 2, 3, 4};
copyNumbers = numbers; - достъпваме същият масив numbers чрез същата референция/път до паметта, но
този път чрез променливата copyNumbers

```

Подмяна на всички елементи

```

String[] namesDevils = sc.nextLine().split(",");
for (int i = 0; i < namesDevils.length; i++) {
    namesDevils[i] = namesDevils[i].replaceAll(" ", "");
}

```

Сортиране

```
Arrays.sort(sumCodes);
```

Попълване на масив с fill

```

public static int[] prevNodes;
for (int i = 0; i < prevNodes.length; i++) {
    prevNodes[i] = -1;
}

Arrays.fill(prevNodes, -1);

```

В Java няма Stream от Characters!!!, или ако има, то не можем да го обърнем в масив от chars

```
Stream<Character> characterStream = testString.chars().mapToObj(c -> (char) c);
IntStream intStream1 = testString.codePoints();
Stream<Character> characterStream2 = testString.codePoints().mapToObj(c -> (char) c);
```

Затова от масив стрингове към масив char правим следното:

```
char[][] matrix = new char[rows][cols];

for (int row = 0; row < rows; row++) {
    String[] tokens = sc.nextLine().split("\\s+"); - масив от стрингове

    for (int col = 0; col < tokens.length; col++) {
        matrix[row][col] = tokens[col].charAt(0); - за всеки стринг елемент, направи го char
    }
}
```

4.3. Печат/Изход на масив

```
String[] numberString = new String[] {"13", "42", "69"};
String joined = String.join("and", numberString); - обединява в 1 стринг масив или колекция от стрингове
```

```
int[] test = {1, 2, 3};
System.out.println(Arrays.toString(test)); - отпечатва без шълкавица - [1, 2, 3]
```

```
int[] numbers;
System.out.println(numbers); - отпечатва шълкавица (16чна бройна система) за разлика от списъците, където разпечатва [6, 6, 3]
```

4.3.1. FOREACH цикъл - масиви / списъци / колекции / – read-only е цикъла

Използва се за обхождане на масиви / листи / колекции

```
for (int number: numbers) {
    System.out.println(number);
}

for (int i = 0; i < numbers.length; i++) {
    int number = numbers[i];
    System.out.println(number);
}
```

5. Методи / Methods

Методите се намират в обект Клас. Когато методите се намират извън обект клас, те се наричат функции. В езиците Java и C# няма функции, а само методи – т.е. всички подобни изчисления са в даден клас под формата на Метод/и.

Return; – приключва изпълнението на дадения метод

Методите в Java са camelCase

Indentation - е разстоянието от началото на лявата страна.

Private – използва се метода само в текущия клас

Public – викаме метод от друг клас в нашия клас

```
static int[] readNextArray(Scanner sc) - връща масив  
static double mathPower(double number, int power) - връща число Double  
static void printInWords(double grade) - връща команда/процес  
static int getMax (int a, int b) - връща число int  
static String repeatString(String s, int repeatCount) - връща стринг
```

Единствено масиви и обекти и колекции, при използването им в метод променят референтната си стойност в RAM паметта, т.е. реално се променя даден елемент от масива. Но при сортиране не работи обаче, не се запаметява сортираната колекция извън метода – защо така!!!

Всички останали типове примитивни данни като

- int, float, double, char, Boolean – стойностен тип данни и
- String – референтен тип данни, но го считаме като стойностен тип тъй като не можем да му сменим съдържанието на String-a.

използвани в даден метод, то метода работи с тяхно копие, и оригиналната им стойност не се променя!!!

Сигнатура на даден метод се състои от **Име на метода и от параметри на метода**.

Типа на метода / на връщаните данни – void, String, int, double, не е част от неговата сигнатура!

Over-loading методи: за да е наличен over-load метод, то метода от един и същи тип (на връщаните данни) и с едно и също име, то неговите параметрите трябва да се различават по поне един от три критерия:

- Брой параметри
- Реда на параметрите
- Типа на параметрите

Или с други думи казано, не може да има два метода с едно и също име, и едни и същи брой, ред и тип параметри, които да връщат различен тип данни/резултат – само 1 такъв метод трябва да съществува и да връща само един тип данни/резултат.

```
static int getMax(int a, int b) {  
    if (a > b) {  
        return a;  
    }  
    return b;  
}
```

Рекурсия единична:

```
static int getMax(int a, int b, int c) {  
    return getMax(getMax(a, b), c);  
}
```

```
static int getMax(String a, int b){  
}
```

```
static int getMax(int b, String a){  
}
```

Използване на скенер в метод

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
  
    inBetween(sc);  
}  
  
private static void inBetween(Scanner sc) {  
    int firstSymbol = sc.nextLine().charAt(0);
```

```

int secondSymbol = sc.nextLine().charAt(0);

for (int i = firstSymbol + 1; i <= secondSymbol - 1 ; i++) {
    System.out.print((char)i + " ");
}

}

```

5.1. Масив и метод

Използването на масив в метод – за да можем да вземем новополучения масив, то трябва да го пишем така:

```

private static int[] firstNElements(int[] numberArr, int count, String evenOrOdds) {
    int[] temp = new int[count];
    .....
    return temp;
}

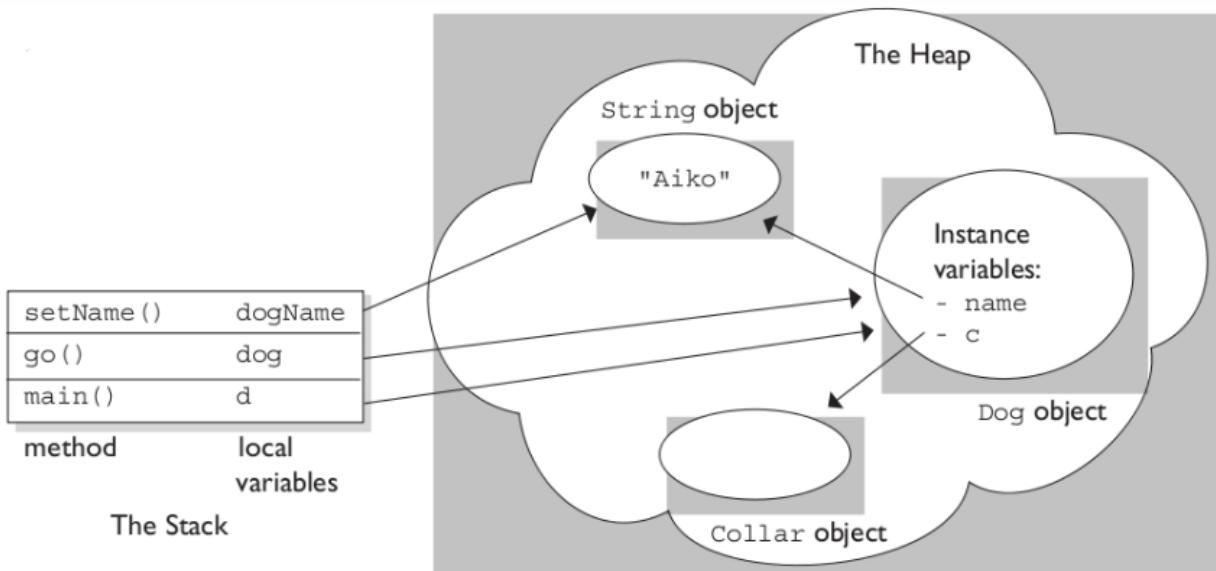
```

5.2. STACK and HEAP - both are in RAM:

STACK - static memory allocation – заделя се при стартирането/компилирането на програмата

HEAP - dynamic memory allocation – определя се / изменя се по време на изпълнение на програмата

Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory . Element of the heap have no dependencies with each other and can always be accessed randomly at any time. You can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.



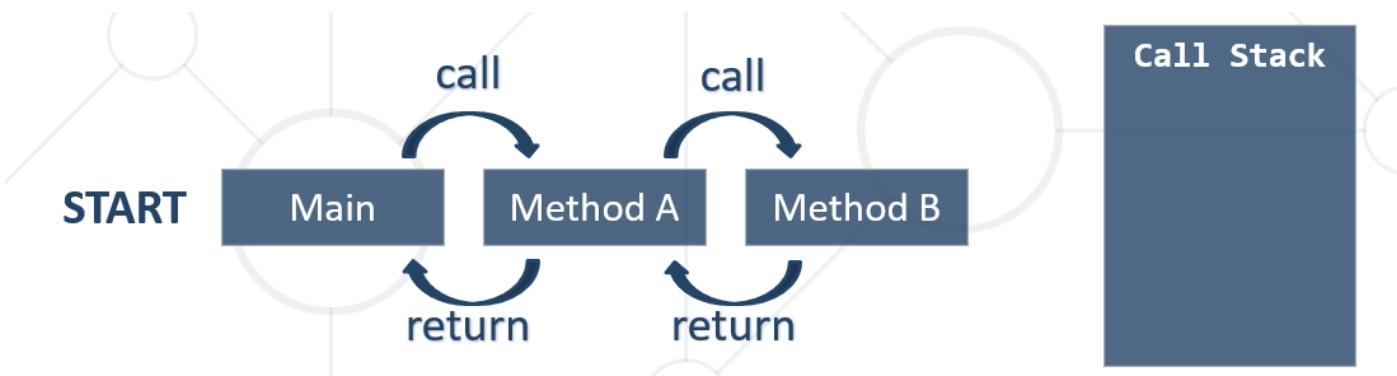
5.3. Debugging when Methods

Stack Frames показва на кое място (първо, второ) в момента на дебъгването/изпълнението на кода се изпълнява даден метод. Комбинацията от методи влиза в състава на Stack-а.

Чрез Frames можем да проследим/проверим по-качествено състоянието на целия стек Stack в даден момент. Ако програмата ни се чупи в даден метод /Frame/, то е твърде вероятно грешката да идва от предходно изпълнения Frame, до който ние имаме достъп!

Един Stack /програма се изпълнява, докато метода main не приключи!

The stack is always reserved in a LIFO order, the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack



6. Списъци / Lists

Без определен брой елементи. Масивите са конкретен вид списъци – с конкретен брой елементи
Списъците работят по-бавно от масивите!

При достигане на дължината на даден лист, то в паметта се заделя памет за 2 пъти вече заделената дължина на списъка

6.1. Четене на лист

```
List<Integer> inputNumbers = new ArrayList<Integer>();
List<Integer> inputNumbers = new ArrayList<Integer>(Arrays.asList(1, 5));
ИЛИ
inputNumbers.add(1);      inputNumbers.add(5);
```

Convert a collection into List

```
List<String> items = Arrays.stream(values.split(" ")).collect(Collectors.toList());
```

Converts an array into list

```
String[] data = sc.nextLine().split(", ");
ArrayList<String> racers = new ArrayList<>(Arrays.asList(data));
```

Converts a list into an array

```
toList().toArray(new Task[size]);
```

Как да трансформирам масив от малък int[] във List<Integer>

Използваме boxed() за вдигане на типа

```
List<Integer> universeSet = Arrays.stream(universe).boxed().collect(Collectors.toList());
```

Като запиша .stream, и автоматично става Arrays.stream()

```
List<Double> items = Arrays.stream(values.split(" ")).map(Double::parseDouble).collect(Collectors.toList());
List<Integer> firstList = Arrays.stream(sc.nextLine().split(" ")).map(Integer::parseInt).collect(Collectors.toList());
```

values e sc.nextLine()

```
List<Integer> list = new ArrayList<>();
for (int i = 0; i < n; i++) {
    int number = Integer.parseInt(sc.nextLine());
    list.add(number);
}
```

Задаване на първоначален размер/капацитет на лист

```
new ArrayList<>(10);
```

6.2. Команди

size() – number of elements in the List<E>

add(element) – adds an element to the List<E>
add(index, element) – inserts an element to given position

След прилагане на remove, дължината на списъка намаля

Remove връща стойността на изтрития елемент

```
int temp = numbers.remove(0);
```

remove(object) – removes by value an element (returns true / false) - ако елементите са тип int, то **Integer.valueOf()** и **стават от големия тип Integer** – премахва само първото срещане на елемента
remove(index) – removes element at index – **от малкия тип int**
List<Integer> numbers = Arrays.stream(sc.nextLine().split(" ")).map(Integer::parseInt).collect(Collectors.toList());
Integer element = Integer.valueOf("5");
numbers.remove(element);

contains(element) – determines whether an element is in the list - ако елементите са тип инт, то **Integer.valueOf()**
set(index, item) – replaces the element at the given index

Добавяне на колекция AdditionalList към колекция initialList

initialList.addAll(лист от елементи additionalList); - допълнителният лист може да го извикаме с метод от тип List<int>

int bombIndex = items.indexOf(bombValue); - връща първият индекс, където се среща стойността bombValue в списъка.
numbers.lastIndexOf(bomb); - връща последният индекс, където се среща стойността

get(i) – вземи i-ият елемент от списъка

Добавяме няколко елемента наведнъж

```
List<List<Integer>> graph = new ArrayList<>();  
graph.add(List.of(3, 6));  
graph.get(3).addAll(Arrays.asList(9, 8, 5)); -
```

Важно – когато работим с Remove, винаги е по-добре да премахнем елемента и запишем във временна променлива; и чак след това да правим други операции със списъка

Също така гледаме дали работим с числа – ако не работим, то можем да използваме String

```
while (number != -1) {  
    inputNumbers.add(number);  
    number = sc.nextInt();  
}  
  
for (int i = inputNumbers.size() - 1; i >= 0; i--) {  
    System.out.print(inputNumbers.get(i) + " ");  
}  
  
for (Integer element : inputNumbers) {  
    System.out.print(element + " ");  
}
```

inputNumbers.add(2, 42); - добавяме нов елемент със стойност 42 на позиция индекс 2, а останалите елементи отиват с една позиция надясно

inputNumbers.set(1, 42); - променяме стойността на елемент на индекс 1, като елемента го променяме на стойност 42

`inputNumbers.remove(1);` - с индекс 1 елемента се маха, и останалите елементи се придвижват с един наляво и общия размер на списъка се намаля с 1 елемент.

Премахване по индекс или премахване по стойност при елементи от `int`

`nums.remove(Integer.valueOf(40));` - премахва **само първият срещнат** елемент/обекта със стойност 40
`nums.remove(1);` - премахва елемента с индекс 1

Премахване на много елементи от лист наведнъж без да правим цикъл:

```
List<Integer> inputNumbers = new ArrayList<Integer>();
```

Създаваме списък от 1 елемент и действаме така:

`inputNumbers.removeAll(new ArrayList<Integer>() {{add(0);}});` - мнимо добавяне на списък-елемент
`inputNumbers.removeAll(new ArrayList<Integer>(Arrays.asList(0)));` - нормално добавяне на списък/елемент

```
Collection<Post> authors = new ArrayList<>(); //можем и така
List<Integer> items = new ArrayList<>();
List<Integer> nums = new ArrayList<>();
for (int i = 0; i < items.size(); i++)
    nums.add(Integer.parseInt(items.get(i)));
if (numbers.size() == 0) е същото като if (numbers.isEmpty())
```

При премахване на елементи

```
List<Integer> numbers = new ArrayList<>(Arrays.asList(2, -2, 20, 42, 39, -1));
int i = 0;
while (i < numbers.size()) {
    if (numbers.get(i) < 0) {
        numbers.remove(i);
    } else {
        i++;
    }
}
```

Когато сравняваме списъци, то винаги е препоръчително да използваме `equals`-дори и когато сравняваме `double` или `int`.

```
merged.add(secondList.get(0));
secondList.remove(0);
```

ИЛИ

```
merged.add(secondList.remove(0)); remove освен, че изтрива елемента, то връща стойността на изтрития елемент.
```

2 списъка в растващ ред да се преобразуват на 1 нов списък пак подреден в растващ ред.

```
while (!firstList.isEmpty() || !secondList.isEmpty()) {
    if (firstList.isEmpty()) {
        merged.add(secondList.get(0));
        secondList.remove(0);
    } else if (secondList.isEmpty()) {
        merged.add(firstList.get(0));
        firstList.remove(0);
    } else {
        if (firstList.get(0) < secondList.get(0)) {
            merged.add(firstList.get(0));
            firstList.remove(0);
        } else {
            merged.add(secondList.remove(0));
        }
    }
}
```

Пример с addAll:

```
1)
List<String> result = new ArrayList<>();
String[] arr = input.split("\\s+");
List<String> listToAdd = Arrays.asList(arr); //от масив към списък
result.addAll(listToAdd);
2)
List<String> result = new ArrayList<>();
String[] arr = input.split("\\s+");
for (String element : arr) {
    result.add(element);
}
```

`Lists<Integer> num = Lists.copyOf(condensed, condensed.size);` - копираме лист с по-къса дължина от началния – създава се нов списък, който сочи към различно място в паметта
`num = condensed;` - num е референция към стойността на същия списък condensed

Сортиране на списъци:

```
Collections.sort(names);
Collections.reverse(names);
Collections.min(numbers);
```

6.3. Печатане/Изход на лист:

```
1) List<String> words = new ArrayList<> (Arrays.asList("the", "quick", "brown", "fox"));
String joined = String.join(", ", words); - обединява в един стринг масив и лист/списък само от стрингове
System.out.println(joined);

2)
private static String joinElementsByDelimeter(List<Integer> items, String delimeter) {
    String output = "";
    for (Integer element : items) {
        output += (element + delimeter);
    }
    return output;
}

String output = joinElementsByDelimeter(numbers, " ");
System.out.println(output);

3) List<Integer> numbers;
System.out.println(numbers); отпечатва се [6, 6, 3]

4) System.out.println(numbers.toString().replaceAll("[\\[\\]]", ""));
- прави от всякакви обекти на стринг, и вече от стринг заменя символите [, или ] с празен интервал/символ
System.out.println(numbers); отпечатва се 6 6 3
```

7. Try Catch конструкция

```
try {
    int integer = Integer.parseInt(input);
} catch (NumberFormatException e){
    isInt = false;
}
```

```

try {
    sum = sumNumbers(arr);
} catch (Throwable th) {
    th.printStackTrace();
    System.out.println(th.getMessage());
}

if (tryParseInt(input)) {
    Integer.parseInt(input); // We now know that it's safe to parse
}

try {
    output += matrix[j].charAt(i);
} catch (Exception e) {
    output += " ";
}

int[] arr = {1, 2, 3};
try {
    System.out.println(arr[3]);
} catch (IndexOutOfBoundsException ex) {
    System.out.println(ex.getMessage());
}

```

7. Рекурсия

Без да принтираме по време на рекурсия

```

public class RecursiveFibonacci {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = Integer.parseInt(sc.nextLine());

        long fib = GetFibonacci(n);
        System.out.println(fib);
    }

    private static long GetFibonacci(int n) {
        if (n == 2 || n == 1) {
            return 1L;
        }

        if (n == 0) {
            return 0;
        }

        return GetFibonacci(n - 1) + GetFibonacci(n - 2);
    }
}

```

С принтиране по време на рекурсия или записване на рекурсията в масив / лист

```

public class TribonacciSequence {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int nNum = Integer.parseInt(sc.nextLine());
        printTrib(nNum);
    }
}

```

```

private static void printTrib(int nnum) { //тук можем да боравим с i-ият елемент от рекурсията
    for (int i = 1; i <= nnum ; i++) {
        System.out.print(tribonacciRecursion(i) + " ");
    }
}

private static int tribonacciRecursion(int num) {
    if (num == 0) {
        return 0;
    } else if (num == 1 || num == 2) {
        return 1;
    } else {
        return tribonacciRecursion(num - 1) + tribonacciRecursion(num - 2) +
tribonacciRecursion(num - 3);
    }
}

```

Дължина:

.size() – за списък

.length – за масив

.length() – за стринг

9. Обекти и класове

Класовете се пишат PascalCase

```

Random rnd = new Random();
rnd.nextInt(5); // [0, 4] 5 е границата и тя не се включва, exclusive
rnd.nextInt(5) + 10; // [10+0, 10+4] в интервала от 10 до 14
nextInt() – връща винаги положително число или нула

```

Вариант с използване на ThreadLocalRandom вместо Random

```
ThreadLocalRandom.current().nextInt();
```

this. – викаме полето(private variable) или метод когато сме в текущия клас

Alt + Insert (Generate) в системата IntelliJ – автоматично настройва:

Constructor

- Getter
- Setter
- Getter and Setter
- Други

Classes define templates for object and consist of:

- Fields (**private variables**) – store values – не е хубаво променливата да се достъпва директно, затова я правим **private**

Следният запис го избягваме!

```

public class Dice {
    public int sides;
}

public class Demo {
    public static void main(String[] args) {
        Dice obj = new Dice();
    }
}

```

```

        obj.sides = 6;
        System.out.println(obj.sides);
    }
}

```

- Getters and Setters – ca public
- Constructors – it is a kind of Setter (Overloading default constructor; Constructor name is the same as the name of the class) – ca public

```

public class Cat {
    private String name;

    public Cat(String name) { //конструктор - overload-ване
        this.name = name;
    }

    public Cat() { //конструктор - overload-ване
    }

    public void setName(String name) { //setter
        this.name = name;
    }

    public String getName() { //getter
        return name;
    }
}

```

- Behaviour - Methods

Objects:

- Hold a set of named values
- Instance of a class - Обектът е инстанция на класа, т.е. е шаблон
- Всеки обект в main метода сочи към някаква референция от паметта

Import ready-to-use packages:

```

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

```

Using *static class members*:

```

LocalDateTime today = LocalDateTime.now();
double cosine = Math.cos(Math.PI);
Integer.parseInt("25");
main винаги е static
Math.abs – статични – не зависи от вътрешното състояние на класа Math
Dice sides6 = Dice.generateWithSides(6); - статичен метод – не зависи от вътрешното състояние на класа Dice

```

Статични полета и методи – могат да бъдат извиквани без да има създадена инстанция на класа

Извикват се само с Името на класа .(точка) името на метода

Нестатични полета и методи – могат да бъдат извиквани само след създаване на инстанция на класа

Using **non-static Java classes**:

```
Random rnd = new Random();
int randomNumber = rnd.nextInt(99);
Dice d = new Dice();      d.setSides(6); - non-static
```

Когато принтираме, самата функция println вика вградения метод на всеки един клас **toString**

```
Articles article = new Articles("", "", "");
System.out.println(article); = System.out.println(article.toString());
```

@Override – казва ни ако сме объркали името на метода **toString**

```
public String toString() {
    String result = String.format("%s - %s:%s", this.title, this.content, this.author);
    return result;
}
```

Пишем Main клас с main метод в package-а – за да може Judge да провери – правим .zip от два файла.

Може да архивирам и целият package на .zip

Другият вариант е да сложим един клас в друг клас и задаваме static – така го предаваме на черния екран

```
List<Person> people = new ArrayList<>();
people
    .stream()
    .filter(p->p.getAge() > 30)
    .sorted((p1, p2) -> p1.getName().compareTo(p2.getName()))
    .forEach(p -> System.out.println(p));
```

```
List<OrderByAge> orderByAgeList = new ArrayList<>();
orderByAgeList
    .stream()
    .sorted((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge())) //сортира възходящо
    .forEach(p -> System.out.println(p));
sorted(Comparator.comparingInt(p -> p.getAge()))
```

1. Можем да добавяме в един клас поле от Тип друг клас – в обекта на инстанцията можем да стигнем до инстанцията на всеки от класовете с подготвени getter-и и точка и точка

```
public class Car {
    private String modelCar;
    private Engine connectWithEnginClassModelEngine;

    public Engine getConnectWithEnginClassModelEngine() {
        return connectWithEnginClassModelEngine;
    }
    public void setConnectWithEnginClassModelEngine(Engine connectWithEnginClassModelEngine) {
        this.connectWithEnginClassModelEngine = connectWithEnginClassModelEngine;
    }

    public class Engine {
        // model, power, displacement and efficiency
        private String modelEngine;
        private String powerEngine;
        private String displacementEngine;
        private String efficiencyEngine;
    }
}
```

В main метода на Main класа:

```
List<Car> carList = new ArrayList<>();
Car currCar = new Car(tokens[0]);
for (Engine engine : engineList) {
    if (engine.getModelEngine().equals(tokens[1])) {
```

```

        currCar.setConnectWithEnginClassModelEngine(engine);
    }
}
carList.add(currCar);

for (Car car : carList) {
    System.out.println(car.getModelCar() + ":" );
    System.out.println(car.getConnectWithEnginClassModelEngine().toString());
    System.out.println(car.toString());
}

```

2. Или можем да имаме поле от тип `List<String>` и да достигаме до него и да добавяме на листа елементи с функцията `add`.

```

public class Team {
    private String teamName;
    private String creatorName;
    private List<String> memberName;

    public Team(String teamName, String creatorName) {
        this.teamName = teamName;
        this.creatorName = creatorName;
        List<String> temp = new ArrayList<>();
        this.memberName = temp;
    }
}

```

В метода `main` – с временна променлива Списък от тип Стинг.

```

Team newTeam = new Team(tokensCreateTeam[1], tokensCreateTeam[0]);
teamList.add(newTeam);
List<String> newMember = new ArrayList<>();
newMember = null;
for (int i = 0; i < teamList.size(); i++) { //Pesho->AiNaBira
    if (tokensJoinMember[1].equals(teamList.get(i).getTeamName())) {
        isTeamCreated = true;
        newMember = teamList.get(i).getMemberName();
        newMember.add(tokensJoinMember[1]);
        teamList.get(i).setMemberName(newMember);
        break;
    }
}

```

Или по този начин:

```

public class Team {
    private String teamName;
    private List<String> membersNames = new ArrayList<>();

    String getMember(int i) {
        return membersNames.get(i);
    }
}

```

В метода `main`:

```

if (tokensCreateTeam[0].equals(team.getMember(0)))

```

10. Associative Arrays (Maps), Lambda and Stream API

10.1. Associative Arrays - Collection of Key and Value Pairs – MAPS

Описание

Hash означава, че има уникални стойности на по-ниско ниво, и не се налага много преобразуване на типове данни

A) **HashMap <key, value>** **HashMap<K, V> - Keys are unique - Uses a hash-table + list**

```
Map<String, Integer> map = new HashMap<>();  
map.put("Ivan", 73);  
map.put("Ivan", 53); - при опит на пре-записване на Иван, се презаписва само стойността от 73 на 53!
```

Произволно записва кой-елемент от листа на кое място е.

Най-бързия и оптимизиран, но не винаги върши работа.

B) **LinkedHashMap <key, value>** **LinkedHashMap<K, V> -Keys are unique -Keeps the keys in order of addition**

```
Map<String, Integer> map = new LinkedHashMap<>();  
Помни последователно клочовете.  
при опит на пре-записване на Иван, се презаписва само стойността от 73 на 53!
```

C) **TreeMap <key, value>** **TreeMap<K, V> Keys are unique Keeps its keys always sorted Uses a balanced search tree(BST) – self-balanced Red-Black tree**

```
Map<String, Integer> map = new TreeMap<>();  
при опит на пре-записване на Иван, се презаписва само стойността от 73 на 53!
```

Въвеждане

```
Map<Integer, Integer> map = new HashMap<>() {  
    put(2, 6);  
};  
int result = map.remove(2); //remove(K) връща стойността V на двойката (K, V)
```

```
char[] input = sc.nextLine().toCharArray();  
LinkedHashMap<Character, Integer> letters = new LinkedHashMap<>();  
for (char letter : input) {  
    letters.putIfAbsent(letter, 0);  
    int count = letters.get(letter);  
    letters.put(letter, count + 1);  
}
```

Прави го immutable collection

```
Map<String, Integer> harvest = Map.of(  
    "Carrots", 0,  
    "Potatos", 0,  
    "Lettuce", 0  
)
```

```
Map<String, List<String>> synonyms = new LinkedHashMap<>();  
List<String> stringList = synonyms.get(key);  
stringList.add(synonym);  
synonyms.put(key, stringList);
```

```
iter/while {  
resources.putIfAbsent(input, 0);  
int oldCount = resources.get(input);  
resources.put(input, oldCount + count);  
}
```

```

double[] numbers = Arrays.stream(sc.nextLine().split("\s+"))
    .mapToDouble(Double::parseDouble)
    .toArray();
Map<Double, Integer> map = new TreeMap<>();
for (double number : numbers) {
    if (!map.containsKey(number)) {
        map.put(number, 1);
    } else {
        map.put(number, map.get(number) + 1);
    }
}

HashMap<String, HashMap<String, Integer>> allPlayers = new HashMap<>();
allPlayers.putIfAbsent(playerName, new HashMap<>());
allPlayers.get(playerName).putIfAbsent(positionSkill, -1);
if (skillPoints > allPlayers.get(playerName).get(positionSkill)) {
    allPlayers.get(playerName).put(positionSkill, skillPoints);
}

```

Методи:

- **put(key, value)** method - airplanes.put("Airbus A320", 150);
- **remove(key)** method - airplanes.remove("Boeing 737");
- **size()** – взема размера
- **containsKey(key)** method - if (map.containsKey("Airbus A320"))

За недублиране на елементи:

```

String[] names = sc.nextLine().split("\s+");
int[] points = Arrays.stream(sc.nextLine().split("\s+"))
    .mapToInt(Integer::parseInt)
    .toArray();
Map<String, Integer> map = new TreeMap<>();

for (int i = 0; i < names.length; i++) {
    String name = names[i];
    int point = points[i];

    if (!map.containsKey(name)) {
        map.put(name, point);
    }
    if (map.containsValue(20)) {
        System.out.println("Already created");
    }
}

```

- **containsValue(value)** method:

```

map.put("Airbus A320", 150);
System.out.println(map.containsValue(150)); //true

```

- **get(K) връща V** – ако ключа го няма, хвърля null (pointer exception) – **иначе връща стойност**
map.get("Ivan") – връща 53.

Не гледа по индекс на ключа, а по стойност на ключа

```

for (String s : map.keySet(name)) {
    if (!map.containsKey(name)) {

```

```
        map.put(name);
    }
}
```

```
Map<String, Integer> test = new HashMap<>();
test.put("a", 1);
test.put("b", 2);
test.put("c", 3);
System.out.println(test.keySet()); - изпечатва всички keys - [a, b, c]
System.out.println(test.values()); - изпечатва всички стойности/value-та - [1, 2, 3]
```

При foreach цикъла, като натиснем iter ни дава предложение дали да създадем:

`for (String s : map.keySet()) {}` - колекция от ключове `map.keySet()` е от тип `Set<String>`
`for (Integer value : map.values()) {}` - колекция от стойности

Обхождане на МАР

Вариант 1 - колекция от двойки ключ и стойност

`items.forEach((k, v) -> System.out.println(String.format("%s -> %d", k, v)));` - когато няма да сортираме и да правим сложни операции

Вариант 2 - чрез iter for цикъл

`for (Map.Entry<String, Integer> keyValuePair : map.entrySet()) {}` - колекция от двойки когато ще правим сортиране и други операции
- entry set - достъпвам двойката key and value заедно, а не поотделно

```
Map<String, Double> fruits = new LinkedHashMap<>();
fruits.put("banana", 2.20);
fruits.put("kiwi", 4.50);
for (Map.Entry<String, Double> keyValuePair : fruits.entrySet()) {
    System.out.printf("%s - %.2f%n", keyValuePair.getKey(), keyValuePair.getValue());
}
```

Вариант 3 - с entrySet и stream API

```
Map<String, List<Integer>> teams = new HashMap<>();
teams.entrySet().stream().....
```

`Set<String>` - уникален лист/списък, в който всички стойности са уникални и не се повтарят

Стойност от тип Лист от Стинг - добавяне на елемент от листа от стринг + ползване на `putIfAbsent()`

`Map<String, List<String>> map = new LinkedHashMap<>(); //`
`map.putIfAbsent(word, new ArrayList<>());` // винаги инициализираме списъка с `new ArrayList()` за да може да добавяме add - този ред се прескача реално ако е в цикъл - *Да избягвам да ползвам putIfAbsent - защото много пъти се налага допълнителна проверка, и по-добре да си пиша if-ве*
`map.get(word).add(syn);`

Пример за ForEach вложен цикъл / вложени мапове

```
LinkedHashMap<String, LinkedHashMap<String, Integer>> contestsAll = new LinkedHashMap<>();
LinkedHashMap<String, Integer> standings = new LinkedHashMap<>();
for (Map.Entry<String, HashMap<String, Integer>> contestName : contestsAll.entrySet()) {
    for (Map.Entry<String, Integer> userName : contestName.getValue().entrySet()) {
        standings.putIfAbsent(userName.getKey(), 0);
        standings.put(userName.getKey(), userName.getValue() + standings.get(userName.getKey()));
    }
}
```

Друг пример за ForEach вложен цикъл / вложени мапове

```
LinkedHashMap<String, LinkedHashMap<String, List<String>>> infoTable = new LinkedHashMap<>();
infoTable.putIfAbsent(continent, new LinkedHashMap<>());
LinkedHashMap<String, List<String>> innerMap = infoTable.get(continent);
innerMap.putIfAbsent(country, new ArrayList<>());
innerMap.get(country).add(city);

for (Map.Entry<String, LinkedHashMap<String, List<String>>> mapEntry : infoTable.entrySet()) {
    String cont = mapEntry.getKey();
    LinkedHashMap<String, List<String>> innerMap = mapEntry.getValue();
    System.out.println(cont + ":");

    for (Map.Entry<String, List<String>> innerMapEntry : innerMap.entrySet()) {
        System.out.println(" " + innerMapEntry.getKey() + " -> " +
                           String.join(", ", innerMapEntry.getValue()));
    }
}
```

Интересни конструкции

```
List<Map<Double, Float>> strange = new ArrayList<>();
List<String>[] arrayOfLists;
```

10.2. Lambda expression – анонимни функции/не си ги пишем ние

```
x -> x / 2      static int func(int x) { return x / 2; }
x -> x != 0      static boolean func(int x) { return x != 0; }
() -> 42         static int func() { return 42; }
```

```
int[] numbers = Arrays.stream(sc.nextLine().split("\s+"))
    .mapToInt(n -> Integer.parseInt(n) / 2)           - това е Lambda израз
    .toArray();
```

Lambda and Stream API help collection processing

.mapToInt(x -> Integer.parseInt(x)) == .mapToInt(Integer::parseInt) е от тип малкия int (IntStream)
.map(x -> Integer.parseInt(x)) == map(Integer::parseInt) е от бащиния тип Integer (Stream<Integer>)

10.3. Stream API / Stream application programming interface

Поток от данни е stream

Lambda and Stream API help collection processing

Първоначалното копие на колекцията не се променя, и това е хубаво при stream

```
Arrays.stream(sc.nextLine().split("\s+"))
    .mapToInt(n -> Integer.parseInt(n) / 2);

.min();
.orElse(2); - или върни 2
.average();
.getAsDouble(); - ако е Optional Min_Max – слагаме накрая и getAsDouble.
```

```
.getAsInt(); - ако е Optional int  
.get() – ако е Optional и го има – така го вземаме  
.sum();  
.findFirst()  
.orElse(0)  
.orElse( null)  
.orElseThrow(() -> new IndexOutOfBoundsException());
```

Интересен вариант:

```
HashMap<String, Department> departments = new HashMap<>();
Department maxAverageSalaryDepartment = departments.entrySet()
    .stream()
    .max((f, s) -> Double.compare(f.getValue().getAverageSalary(), s.getValue().getAverageSalary()))
    .get()
    .getValue(); или getKey();
```

Mapping

map() - manipulates elements in a collection:

`.mapToInt(p -> Integer.parseInt(p))` което е същото като `.mapToInt(Integer::parseInt())`

Правим/презписваме нов масив/лист със същото име, но с филтрирани примерно елементи

```
String[] words = {"abc", "def", "geh", "yyy"};
List<String> words = {"abc", "def", "geh", "yyy"};
words = Arrays.stream(words)
    .map(w -> w + "yyy")
    .toArray(String[]::new); // за масив от бащин тип / wrap класа на типа
    .toArray(Character[]::new); // за масив
    .collect(Collectors.toList()); // за колекция вид List
```

`.toArray():` - прави го на масив

.collect(Collectors.toList()); - прави го на List или на друг вид колекция, например на Map.Entry
.collect(Collectors.toSet()); - прави го на сет, т.е. само с уникални стойности

`.collect(Collectors.toCollection(LinkedHashSet::new));` - прави го на колекция, която ние си искаме, в случая искаме LinkedHashSet

`.collect(Collectors.toCollection(ArrayDeque::new));` - вариант за връщане на `ArrayDeque`

```
private static LinkedHashSet<Integer> readDeck(String nextLine) {  
    return Arrays.stream(nextLine.split("\\s+"))  
        .map(Integer::parseInt)  
        .collect(Collectors.toCollection(LinkedHashSet::new))  
}
```

`.stream()`
`.map(c-> c.toString())` – прави го от число примерно на стринг
`.collect(Collectors.joining(", "));` - прави stream-а на String, със запетая и space обединен

```
String result = Arrays.stream(sc.nextLine().split("\\s+"))
    .map(n -> Integer.parseInt(n))
    .sorted((a, b) -> b.compareTo(a))
    .limit(3)
```

```

.map(n -> n.toString())
.collect(Collectors.joining(" "));

System.out.println(result);

```

От лист от Integer го правим на лист от Стингове, които в последствие обединяваме по space

```

List<Integer> arr = Arrays.stream(sc.nextLine().split("\\s+")).map(Integer::parseInt)
    .collect(Collectors.toList());

```

```

System.out.println(arr.stream()
    .map(Object::toString)
    .collect(Collectors.joining(" ")));

```

Filtering

```

.filter(n -> n > 0);
.filter(w -> w.length() % 2 == 0)

```

```

int min = Arrays.stream(new int[]{15, 25, 35}).min().getAsInt();
int max = nums.stream().mapToInt(Integer::intValue).max().getAsInt();
int max = nums.stream().max(Integer::compareTo).get();
int sum = nums.stream().mapToInt(Integer::intValue).sum();

```

Ordering/Sorting

```

.sorted((n1, n2) -> n1.compareTo(n2)) - ascending
.sorted((n1, n2) -> n2.compareTo(n1)) - descending
.limit(3) - ограничи сортировката до първите 3 стойности

```

.sort() – е на масив / Лист, но не и на поток данни stream

Sorting Collections by Multiple Criteria

a.compareTo(b) – когато е бащин Integer, Double или String или Character
Double.compare(a, b)
Integer.compare(second, first) или second - first; - когато е само int

```

.sorted((e1, e2) -> {
    int res = e2.getValue().compareTo(e1.getValue()); // сортираме по Value – compareTo() връща 1, 0 или -1
    if (res == 0)
        res = e1.getKey().compareTo(e2.getKey()); //ако са равни, то ги сортираме и по Key
    return res; })
.forEach(e -> System.out.println(e.getKey() + " " + e.getValue()));

```

```

Map<String, List<Integer>> teams = new HashMap<>();
teams.put("Sanow", Arrays.asList(1, 23, 45));
teams.put("Sbnow", Arrays.asList(19, 39, 29));
teams.put("Acb", Arrays.asList(45, 23, 12));

teams.entrySet()

```

```

.stream()
.sorted((e1, e2) -> {
    if (e1.getKey().compareTo(e2.getKey()) == 0) {
        int sum1 = e1.getValue().stream().mapToInt(x ->
Integer.parseInt(x+"")).sum();
        int sum2 = e1.getValue().stream().mapToInt(x ->
Integer.parseInt(x+"")).sum();

        return sum1 - sum2; //Връща 1 0 или -1
    }
    return e2.getKey().compareTo(e1.getKey());
})
.forEach(e -> { //functional forEach
    System.out.println("Key :" + e.getKey());
    System.out.println("Values -> ");
    e.getValue().sort(Integer::compare);
    for (Integer age : e.getValue()) {
        System.out.printf("---%d%n", age);
    }
});
```

Ordering/Sorting with Reverse

```

List<Integer> numbers = Arrays.stream(sc.nextLine().split("\s+"))
    .map(Integer::parseInt)
    .sorted(Collections.reverseOrder()) - по възходящ/низходящ ред ги прави
    .collect(Collectors.toList());
```

Междинно достъпване на елементите в Stream-a

```

e.stream()
.peek(z-> System.out.print(z + " "));
```

Out/printing

Можем и с итерация / for цикъл `entrySet()` или `.keySet` или `.valueSet`

Вариант 1 – с двойка ключ и стойност

```

resources
    .forEach((k, v) -> System.out.println(String.format("%s -> %d", k, v)));
```

Вариант 2 – чрез iter for цикъл

`for (Map.Entry<String, Integer> keyValuePair : map.entrySet()) {}` – колекция от двойки когато ще правим сортиране и други операции

- `entry set` – достъпвам двойката key and value заедно, а не поотделно

```

Map<String, Double> fruits = new LinkedHashMap<>();
fruits.put("banana", 2.20);
fruits.put("kiwi", 4.50);
for (Map.Entry<String, Double> keyValuePair : fruits.entrySet()) {
    System.out.printf("%s - %.2f%n", keyValuePair.getKey(), keyValuePair.getValue());
}
```

Вариант 3 – с entrySet и API stream

```

items
    .entrySet()
    .stream()
    .sorted((i1, i2) -> i2.getValue() - i1.getValue())
//.sorted((i1, i2) -> i2.getValue().compareTo(i1.getValue()))
```

```

.forEach(i -> System.out.println(String.format("%s: %d", i.getKey(), i.getValue())));

letters
    .entrySet() или .keySet или .valueSet
    .forEach(p -> System.out.println(String.format("%c -> %d", p.getKey(), p.getValue())));

courses
    .entrySet()
    .stream()
    .sorted((c1, c2) -> {
        int first = c1.getValue().size();
        int second = c2.getValue().size();
        return Integer.compare(second, first);
    })
    .forEach(c -> {
        System.out.println(String.format("%s: %d",
            c.getKey(),
            c.getValue().size()));

        c.getValue()
            .stream()
            .sorted((s1, s2) -> s1.compareTo(s2))
            .forEach(s -> System.out.println(String.format("-- %s", s)));
    });

```

Използване на final AtomicInteger и AtomicReference<Integer>

```

final AtomicInteger br = new AtomicInteger();
В stream-а във forEach частта при разпечатване:
.forEach(s -> {
    System.out.println(String.format("%s. %s <::> %d", br.incrementAndGet(), s.getKey(),
    s.getValue()));
});
br.set(1);

```

Също така става и с масив от 1 елемент със стойност 0 - РАБОТИ ☺

```

public String getAllMinutesPlaylistSongs() {
    Integer[] sumSecond = new Integer[1];
    sumSecond[0] = 0;
    this.playlistRepositorySongsUsers.findAll()
        .forEach(s -> {
            sumSecond[0] += s.getDuration();
        });

    return sumSecond[0].toString();
}

```

Също така става и с обикновена променлива, която не променя стойността си от създаването си до използването ѝ в stream-а, т.е. е final - РАБОТИ ☺

```
AtomicReference<Integer> atomicReference = new AtomicReference<>();
```

```
.forEach(contnt -> {
    System.out.println(String.format("%s: %d participants", contnt.getKey(),
    contnt.getValue().size()));
```

```
Map<String, Integer> students = new HashMap<>();
students = contnt.getValue();
students
    .entrySet()
    .stream()
    .sorted((.....
})
```

11. Text Processing

Strings are immutable (read-only)

Accessible by index (read-only)

Strings use Unicode

11.1. Initializing a String

```
String str = "Hello, Java";
String name = sc.nextLine();

String name = new String("Pesho");
```

Converting a **string** from and to a **char array**:

```
String str = new String(new char[] {'s', 't', 'r'});
char[] charArr = str.toCharArray();
```

!!! `String.valueOf(5)` е същото като `(5 + "")`

ПРАВИЛО: - ако трябва да записваме нова стойност в стринга, да проверяваме дали трябва да дадем `text = text.substring(...)` примерно

11.2. Manipulating Strings using the String Class—**масив от Char**, по-малко на брой операции позволява

Concatenating

```
String text = "Hello" + ", " + "world!";
```

```
String text = "Hello, ";
text += "John";
```

Use the `concat()` method

```
String greet = "Hello, ";
String name = "John";
String result = greet.concat(name); // "Hello, John"
```

Join

Joining Strings

`String.join("", ...)` concatenates strings

```
String t = String.join("", "con", "ca", "ten", "ate");
```

Joining Strings - an array/list of strings

```
String s = "abc";
String[] arr = new String[3];
for (int i = 0; i < arr.length; i++) { arr[i] = s; }
String repeated = String.join("", arr); // "abcabcabc"
```

Или по този начин с `Collectors.joining(", ")`

```
String[] words = sc.nextLine().split(" ");
String streamResult = Arrays.stream(words)
```

```
.map(w -> repeatTimes(w))
.collect(Collectors.joining(", "));
```

Substring

```
substring(int startIndex, int endIndex)
String card = "10C";
String power = card.substring(0, 2); - взема до endIndex минус 1
System.out.println(power); // 10
```

```
substring(int startIndex)
String text = "My name is John";
String extractWord = text.substring(11);
System.out.println(extractWord); // John
```

Searching – indexOf(), lastIndexOf(), contains()

indexOf() - returns the first match index or -1

```
String fruits = "banana, apple, kiwi, banana, apple";
System.out.println(fruits.indexOf("banana")); // 0
System.out.println(fruits.indexOf("orange")); // -1
```

lastIndexOf() - finds the last occurrence

```
String fruits = "banana, apple, kiwi, banana, apple";
System.out.println(fruits.lastIndexOf("banana")); // 21
System.out.println(fruits.lastIndexOf("orange")); // -1
```

contains() - checks whether one string contains another

```
String text = "I love fruits.";
System.out.println(text.contains("fruits")); // true
System.out.println(text.contains("banana")); // false
```

endsWith() - checks whether one string ends with another

```
if (email.endsWith(".uk"))
```

Replacing

С използването на **indexOf** и **substring**

```
String toRemove = sc.nextLine();
String text = sc.nextLine();

while (text.contains(toRemove)) {
    int toRemoveIndex = text.indexOf(toRemove);
    int toRemoveLength = toRemove.length();

    text = text.substring(0, toRemoveIndex) +
        text.substring(toRemoveIndex + toRemoveLength);
```

ИЛИ

```
replace(match, replacement) - replaces all occurrences - работи с RegEx
String toRemove = sc.nextLine();
String text = sc.nextLine();
text = text.replace(toRemove, "");
```

Splitting

Split a string by given pattern

```
String[] words = text.split(", ");
```

```
Split a string by given pattern into n numbers of elements
String[] tokens = sc.nextLine().split("\\s+", 2); - връща 2 елемента
String[] tokens = sc.nextLine().split("\\s+")[0]; - след split-а, взема първият елемент
```

```
Split by multiple separators
String text = "Hello, I am John.";
String[] words = text.split("[, .]+");
// разделя по който и да е от регулярните изрази(regular expression)
// "Hello", "I", "am", "John"
```

Прохождане на String-а / масива от чарове

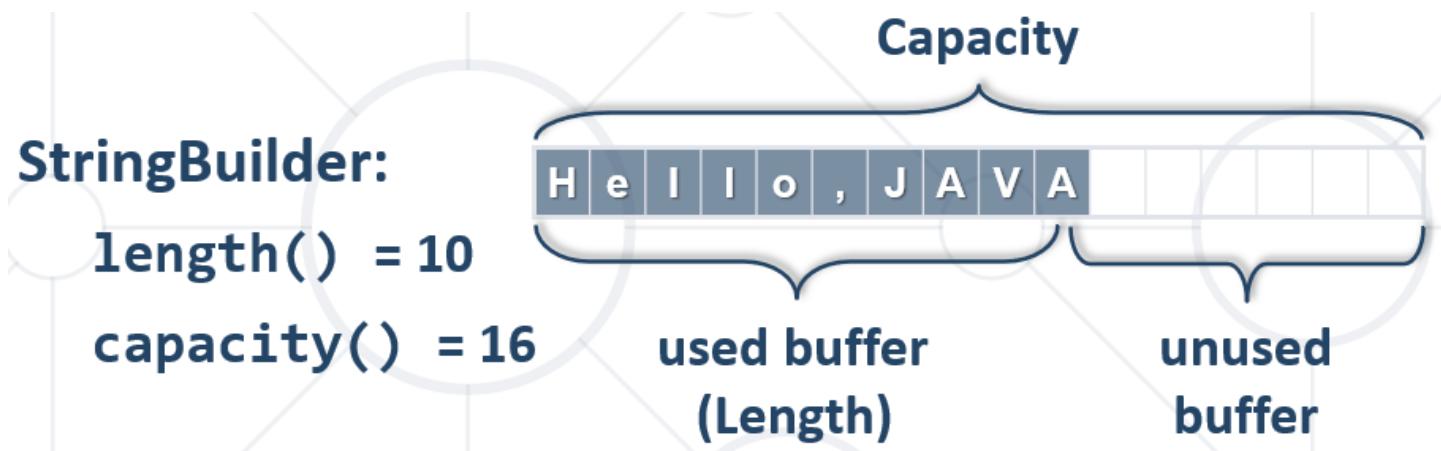
```
String message = sc.nextLine();
for (int i = 0; i < message.length(); i++) {
    char letter = message.charAt(i);
}
```

ПРАВИЛО: - ако трябва да записваме нова стойност в стринга, да проверяваме дали трябва да дадем
text = text.substring()... примерно

11.2. Using the StringBuilder Class – списък от Char – с бонус операции в сравнение работа с нормалния String

StringBuilder keeps a buffer space, allocated in advance and **it works a lot quicker**

Concatenating strings is a **slow** operation because each iteration **creates a new string**



Инициализация или от Стринг в StringBuilder

```
StringBuilder sb = new StringBuilder("Hello,");
    sb.append("John! ");
    sb.append("I sent you an email.");
System.out.println(sb.toString()); // Hello, John! I sent you an email.
```

append()

- appends the string representation of the argument

```
StringBuilder sb = new StringBuilder();
sb.append("Hello Peter, how are you?");
```

```
comments.append(String.format("<div>%n    %s%n</div>%n", input));
```

System.out.println(comments.toString()); - печата форматираната версия, с нови редови и т.н.

```
length()
- holds the length of the string in the buffer
System.out.println(sb.length()); // 25
```

setLength(0) – скъсява дължината на стринга
- removes all characters -това е равно в класа String на String result = "";

setCharAt(int index, char ch); - на кой индекс да сменим символа

```
charAt(int index)
- returns char on index
StringBuilder sb = new StringBuilder();
sb.append("Hello Peter, how are you?");
System.out.println(sb.charAt(1)); //e
```

```
insert(int index, String str)
- inserts a string at the specified character position
sb.insert(11, " Ivanov");
System.out.println(sb); // Hello Peter Ivanov, how are you?
```

indexOf() - returns the first match index or -1
lastIndexOf() - finds the last occurrence

```
replace(int startIndex, int endIndex, String str)
- replaces the chars in a substring - без присвояване работи sb = sb.replace
StringBuilder sb = new StringBuilder();
sb.append("Hello Peter, how are you?");
sb.replace(6, 11, "George"); - премахни нещата от 6ти до 11ти индекс, и на тяхно място сложи George
//Hello George, how are you?"
```

toString() – изход/печат
- converts the value of this instance to a String - преобразува масива от данни StringBuilder
отново на String
String text = sb.toString();
System.out.println(text); // "Hello George, how are you?"

delete()
StringBuilder letters = new StringBuilder();
letters = letters.delete(0, 4); - изтрива от 0ия индекс до 3тия индекс (десния индекс минус 1)

insert()
letters = letters.insert(2, "pesho");

substring()
String sub = **substring**(int startIndex, int endIndex); - взема до endIndex минус 1
String sub = **substring**(int startIndex)

reverse()
StringBuilder sb = new StringBuilder();
System.out.println(sb.reverse().toString());
String reversed = new StringBuilder("ABC").reverse().toString();

```
Chain concatenating / building string
public static StringBuilder out = new StringBuilder();
out.append("Step #").append(steps++).append(": Moved disk").append(System.LineSeparator());
```

11.3. Using the Character Class

```
Character.isDigit(5) - връща true
Character.isLetter('z'); - връща true
Character.isLetterOrDigit('z');
Character.isUppercase
```

12. Regular Expressions (RegEx)

Match text by pattern

<https://regex101.com/>

<https://regexr.com/>

<http://regexone.com>

"|" Or (или)

За група от символи:

[nvj] matches any character that is either n, v or j

[abc][def] първи символ а б или с; втори символ д, е или f

[^a] matches any character that is not a

[^abc] matches any character that is not a, b or c

[0-9] character range matches any digit from 0 to 9

[0-9]+ matches non-empty sequence of digits – повече от една цифра последователно записана

[A-Z][a-z]* matches a capital for sure + small letters

. matches any character everything except for line terminators

\. – символът точка само – ескейпваме специалния знак точка

[A-Z][a-z]+ [A-Z][a-z]+ точно една главна буква с повече малки букви плюс space плюс още един път една главна буква с повече малки букви

John Smith

- \w - matches any **word character** (a-z, A-Z, 0-9, _) == [A-Za-z0-9_] - дефиниция за **alphanumeric**
- \W - matches any **non-word character** (the opposite of \w)
- \s - matches any **white-space character**
- \S - matches any **non-white-space character** (opposite of \s)
- \d - matches any **decimal digit** (0-9)
- \D - matches any **non-decimal character** (the opposite of \d)

- \b To prevent capturing of letters across new lines, put "\b" at the beginning and at the end of your regex
- работи за думи в изречение, когато не гледаме за начало и край на ред с ^(\$|am (Svilen))\$
- matches the empty string at the beginning or end of a word

При завършване на дума/изречение с точка, въпросителен, удивителен, запетая – го взема

При \bFoo – гледа дали преди Foo е точка, запетая, space, удивителен, въпросителен

При Foo\b – гледа дали след Foo е точка, запетая, space, удивителен, въпросителен

\b is zero-width, it doesn't actually match any character

Quantifiers

* matches the previous element **zero or more times** – мачва колкото се може повече цифри d

\+\d* +**359885976002 a+b**

+ matches the previous element **at least ONE or more times** - мачва колкото се може повече цифри d

\+\d+ +**359885976002 a+b**

? matches the previous element **zero or one time** – мачва до една цифра d

\+\d? +**359885976002 a+b**

{3} matches the previous element exactly 3 times – мачва до 3 цифри d

\+\d{3} +**359885976002 a+b**

{3,8} matches the previous element exactly minimum 3 times – мачва 3 или повече цифри d

\+\d{3,8} +**359885976002 a+b**

{3,10} matches the previous element exactly minimum 3 to 10 times – трябва да има поне 3 съвпадение или до 10 съвпадения цифри d

\+\d{3,11} +**359885976002 a+b**

? – **optionality** ab?**c** will match either the strings "abc" or "ac" because the b is considered optional

\? plain character ?

\(.+?\) – когато имаме отваряща и затваряща скоба, използваме \ защото иначе го разпознава като група.

.*? matches between zero and unlimited times (**lazy model – as few times as possible**, до **първото срещане на backreference** например)

.* matches between zero and unlimited times (**greedy model – as more times as possible**, до **последното срещане на backreference** например)

(**subexpression**) - captures the matched subexpression as numbered group – ограждаме в скоби

(?:**subexpression**) - defines a non-capturing group – група, която не се брои - **The parser uses it to match the text, but ignores it later, in the final result.**

(?<name>**subexpression**) - defines a named capturing group – ограждаме в скоби и си слагаме име на всяка група

(?<day>\d{2})-(?<month>\w{3})-(?<year>\d{4}) **22-Jan-2015**

Шаблон за е-мейл – как изглеждат данните, които търся

\w+@[A-Za-z]+\.[A-Za-z]+

valid123@email.bg

invalid*name@email1.bg

the start and the end of the line using the special ^ and \$

^(I) am (Svilen)\$ - изразът трябва да започне с I и да завърши с Svilen

Note that this is different than the hat used inside a set of bracket [^...] for excluding characters, which can be confusing when reading regular expressions.

Capture as a group

Правим проверка на всичко, но хващаме в група само това, което е преди точката и разширението ^(.+)\.pdf\$ - всички имена до точката, т.е. без точката и разширението на файла

Nested groups / capture subgroup

(\w+ (\d+)) – хванатите групи на Jan 1987 са Jan 1987 и 1987

I love (cats|dogs) – regex котки или кучета

I love cats

I love dogs

Backreferences

\number - matches the value of a numbered capture group

<(\w+)[^>]*>.*?<\1>

Вземи <

след това вземи една или няколко букви без >

след това вземи >

след това вземи който и да е символ много на брой пъти, но lazy model – при първото срещане на backreference .*?

след това вземи </

след това вземи референция от група 1, може да вземаме от други групи референции

накрая сложи >

Regular Expressions are cool!

<p>I am a paragraph</p> ... some text after

Hello, <div>I am a<code>DIV</code></div>!

Hello, I am Span

SoftUni

Regex in Java library

Basic declarations/operations

java.util.regex.Pattern

java.util.regex.Matcher

```
Pattern pattern = Pattern.compile("a*b");
Matcher matcher = pattern.matcher("aaaab");
```

```
boolean match = matcher.find();           - searches for the next match
String matchText = matcher.group();        - gets the matched text
```

```
System.out.println(matcher.groupCount()); - показва колко групи има мащнати от шаблона / групите
на шаблона в обикновени скоби ()
```

```
String text = "Andy: 123";
String pattern = "([A-Z][a-z]+): (?<number>\\d+)";
```

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

```

Matcher matcher = regex.matcher(text);

System.out.println(matcher.find());           // true !!! - като дадем веднъж find, 2-ри път не
може да го намери, освен ако не ресетнем matcher-а
matcher.group() - съответства на намереното съвпадение
System.out.println(matcher.group());          // Andy: 123 - всичко
System.out.println(matcher.group(0));          // Andy: 123 - всичко
System.out.println(matcher.group(1));          // Andy - първа група
System.out.println(matcher.group(2));          // 123 - втора група
System.out.println(matcher.group("number"));    // 123 - група с име number

```

```

Pattern pat = Pattern.compile("\b[A-Z][a-z_]+ [A-Z][a-z]+");
Matcher matcher = pat.matcher(text);
while / if (matcher.find()) {
    System.out.print(matcher.group(0) + " ");
}

```

matcher.reset(); - дори и да сме намерили съвпадение (то казваме, че не сме намерили), то почни да търсиш наново по първото съвпадение (има логика ако сменим регекса на matcher-а)

String a = "\\\\"; - това е само символът \

Използване за шаблон String.format с променливи стойности

```

String pattern3 = String.format("^(%c[^ ]{%d})$", letter.getKey(), letter.getValue() - 1);
Pattern word3Pattern = Pattern.compile(pattern3);
for (int i = 0; i < word3.length-1; i++) {
    Matcher word3Matcher = word3Pattern.matcher(word3[i]);
    if (word3Matcher.find())

```

Replacing With Regex

To replace **every/first** subsequence of the input sequence that matches the pattern with the given replacement string

replaceAll(String replacement)
replaceFirst(String replacement)

```

String regex = "[A-Za-z]+";
String string = "Hello Java";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(string); - резултата от прилагането на
String res = matcher.replaceAll("hi");    // hi hi
String res2 = matcher.replaceFirst("hi"); // hi Java

```

Splitting with Regex

split(String pattern) - splits the text by the pattern

```

String text = "1 2 3      4";
String pattern = "\s+"; - отбележва което и да е от четирите вида whitespace
String[] tokens = text.split(pattern);

```

Whitespaces:

space (_),
tab (\t)
new line (\n)
the carriage return (\r)

\s will match **any** of the specific whitespaces above

Цяло число или дробно число

"(-?\d*\.\d+)"

```
String regex = "^(|\s)[a-zA-Z][\\._\\-a-zA-Z]*[a-zA-Z]@[a-zA-Z][\\._a-zA-Z]*[a-zA-Z]\\.[a-zA-Z]{2,}";  
Pattern pattern = Pattern.compile(regex);  
Matcher matcher = pattern.matcher(input);  
while (matcher.find()) {  
    sb.append(matcher.group() + "\n");  
}
```

Какво прави дългия регекс:

^|\s - Искаме да проверим дали има начало на стринг ИЛИ (" | ") дали има празно място (това прави първа група)

[a-zA-Z] - една малка буква или число

[\\._\\-a-zA-Z]* - дали са точка, долна черта, тире, малка буква или число

[a-zA-Z] - една малка буква или число

@ - маймунка

[a-zA-Z] - една малка буква или число

[\\._a-zA-Z]* - дали са точка, тире, малка буква или число

[a-zA-Z] - една малка буква или число

\\. - точка

[a-zA-Z]{2,} - две или повече малки букви

Alt + Enter върху регекс израза в Java – имаме опция да проверяваме в самата Java дали даден израз отговаря / се match-ва.

<https://www.regular-expressions.info/refadv.html> - Regular Expression Reference: Special Groups
positive lookahead и negative lookahead

13. Stack – LIFO (Last In, First Out)

Като цяло гледаме с дебъгване това, което създаваме дали е това което искаме като структура от данни
Стекът е линейна структурата от данни в информатиката, в която обработката на информация става само от едната страна наречена **връх**. **Дъното** не е и не трябва да е достъпно. Стековете са базирани на принципа „последен влязъл пръв излязъл“ (от английски: *LIFO – Last In First Out*)



Stack<Integer> stack = new Stack<>(); - това не го ползваме, няма функционалности за поддръжка, за стари процесори, които вече ги няма

Връх на стек – последния добавен елемент!

Да Използваме `ArrayDeque` или само като стек, или само като опашка.

Методите в `CallStack` работят на принципа на `Stack`

Creating Stack

0 index – the Peak	1 index	2 index	3 index – the Bottom
4 th time push	3 rd time push	2 nd time push	1 st time push

`ArrayDeque<Integer> stack = new ArrayDeque<>();` - Creating a Stack

Можем да добавяме елементи както в началото, така и в края на стека. Но добавяме в началото, на индекс 0
Данните на `ArrayDeque` се съхраняват в най-бързата оперативна памет – именно Cache Паметта на процесора.

Чете ги като стек

```
ArrayDeque<String> stack = new ArrayDeque<>();
Arrays.stream(sc.nextLine().split("\\s+")).forEach(e -> stack.push(e));
При Mimi Pepi Toshko
```

0 – the Peak	1 index	2 index – the Bottom
Toshko	Pepi	Mimi

Чете ги като стек по обикновения начин

```
String[] children = scanner.nextLine().split("\\s+");
ArrayDeque<String> stack = new ArrayDeque<>();
for (int i = children.length - 1; i >= 0; i--) {
    String child = children[i];
    stack.offer(child);
}
for (String child : children) {
    stack.push(child);
}
```

// добавяме колекцията `tokens` в празната колекция `stack` – получава се обърната редица
`String[] tokens = sc.nextLine().split("\\s+");`
`Deque<String> stack = new ArrayDeque<>();`
`Collections.addAll(stack, tokens);` // добавяме колекцията `tokens` в празната колекция `stack`

Operations

`stack.push(element);` - Adding elements at the top, the Peak – добавяме елемент на индекс 0, а останалите елементи се преместват надясно (с 1 индекс плюс)

`stack.add(element);` - добавям елемент накрая, като последен индекс.

`Integer element = stack.pop();` - Removing element at the top/the Peak and returning its value – премахваме елемент от индекс 0 и връщаме стойността му; - при изтриване на елемент, не се прави операция по преместване на елементите

`stack.remove(object);` - можем да премахнем който и да е елемент то стека, но това е не концепцията за стек

`Integer element = stack.peek();` - Getting the value of the topmost element, which is at index 0

```
public static Deque<Integer> source = new ArrayDeque<>();
public static Deque<Integer> destination = new ArrayDeque<>();
destination.push(source.pop());
```

```

int size = stack.size(); - размерът

boolean isEmpty = stack.isEmpty(); // stack.size() == 0
stack.isBlank() == null или == "" от Java 11 и нагоре. Judge системата работи с до Java 10.

boolean exists = stack.contains(2) - дали го има зададения елемент

Collections.min(numbers);

element = stack.clear(); - Clearing the stack

```

Сортиране – минаваме през друга структура от данни:

```

ArrayDeque<String> stack_IDs_available = new ArrayDeque<>();
Arrays.stream(sc.nextLine().split("\\s+")).forEach(e -> stack_IDs_available.push(e));
//sort ascending
List<String> collectSorted = stack_IDs_available.stream().sorted().collect(Collectors.toList());
stack_IDs_available.clear();
collectSorted.stream().forEach(e -> stack_IDs_available.push(e)); ///add in a stack again

```

Печатане/Изход

```

ArrayDeque<String> stackMy = new ArrayDeque<>();
System.out.println(stackMy.size()); - изпечатва размера на стека / броят елементи

```

Можем да итерираме елементите от Stack с foreach цикъл, но не можем да го обходим с нормален for цикъл. Т.е. нямаме достъп до индексите на стека.

```

for (Integer elem : stack) {
    System.out.print(elem);
}

while (!stackNumbers.isEmpty()) {
    System.out.print(stackNumbers.pop() + " ");
}

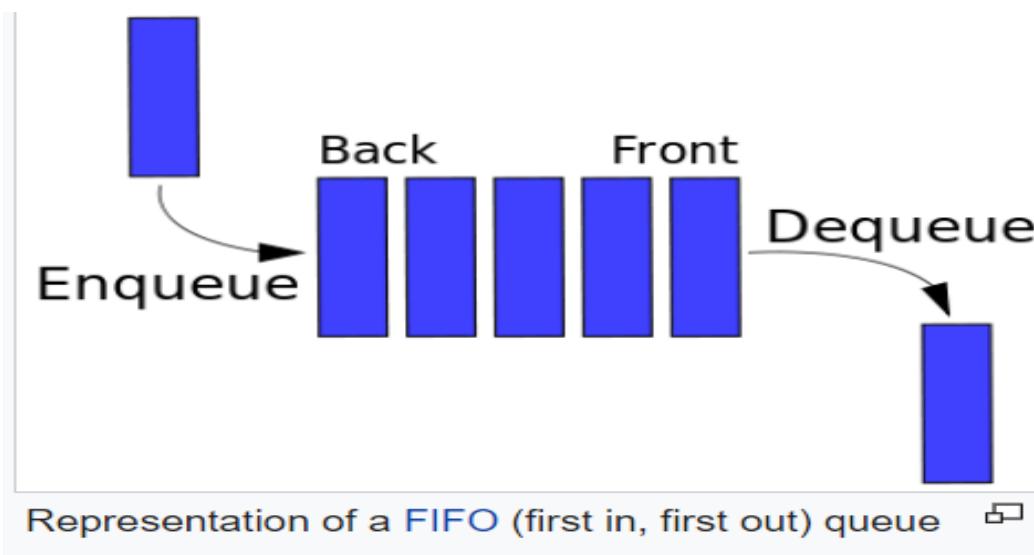
```

14. Queue / Опашка – FIFO (First In, First Out)

Като цяло гледаме с дебъгване това, което създаваме дали е това което искаме като структура от данни

Опашката представлява крайно, линейно множество от елементи, при което елементи се добавят само най-отзад (enqueue) и се извличат само най-отпред (dequeue). Абстрактната структура опашка изпълнява условието "първият влязъл първи излиза" (FIFO: First-In-First-Out). Това означава, че след като е добавен един елемент в края на опашката, той ще може да бъде извлечен (премахнат) единствено след като бъдат премахнати всички елементи преди него в реда, в който са добавени.

Структурата опашка и поведението на нейните елементи произхождат от ежедневната човешка дейност. Например опашка от хора, чакащи на каса за билети. Опашката има **начало (the head or front)** и **край (back, tail, or rear of the queue)**. Новодошлите хора застават последни на опашката и изчакват докато постепенно се придвижват към началото. По този начин опашката изпълнява **функцията на буфер**.



`Vector<Integer> vector = new Vector<>();` - това не го ползваме, няма функционалности за поддръжка, за стари процесори които вече ги няма

Да използваме `ArrayDeque` или само като стек, или само като опашка.

Creating a Queue

0 – the Front	1 index	2 index	3 index – the Back
1 st time offer	2 nd time offer	3 rd time offer	4 th time offer

`ArrayDeque<Integer> queue = new ArrayDeque<>();` - Creating a Queue

Можем да добавяме елементи както в началото (индекс 0), така и в края на опашката (`.size() - 1`). Но добавяме в края, като последен елемент с последен най-голям десен индекс

Данните на `ArrayDeque` се съхраняват в най-бързата оперативна памет – именно Cache Паметта на процесора.

Чете ги като опашка

```
ArrayDeque<String> queue = Arrays.stream(sc.nextLine().split("\\"s+"))
    .collect(Collectors.toCollection(ArrayDeque::new));
```

При Mimi Pepi Toshko

0 – the Front	1 index	2 index – The Back
Mimi	Pepi	Toshko

Чете ги като опашка по обикновения начин

```
String[] children = scanner.nextLine().split("\\"s+");
ArrayDeque<String> queue = new ArrayDeque<>();
for (String child : children) {
    queue.offer(child);
}
```

Operations

Add

`queue.add(element);` - throws exception if queue is full

`queue.offer(element);` - returns false if queue is full – това да използвам за добавяне на елемент на опашка – елемента се добавя накрая, като краен/последен, с последен най-голям десен индекс/The Back

`offer()` – returns `false` if queue is full

`add()` – throws exception if queue is full

```
for (String child : children)
    queue.offer(child);
```

```
String[] children = sc.nextLine().split(" ");
ArrayDeque<String> queue = new ArrayDeque<>();
Collections.addAll(queue, children); - Правим масива на опашка – копира колекция children в
колекция queue
```

Removing elements:

`element = queue.remove();` - throws exception if queue is empty

`element = queue.poll();` - returns null if queue is empty - **това да използвам за изтриване на елемент на опашка – премахва елемент от началото на опашката (индекс 0), the Front** – реално ползва `removeFirst()`

`poll()` - returns **null** if queue is empty

`remove()` - throws exception if queue is empty

`element = queue.peek();` - **Getting the value of the topmost first element, which is at index 0, the Front**

Utility Methods

```
Integer element = queue.peek(); - checks the value of the first element
Integer size = queue.size(); - returns queue size
Integer[] arr = queue.toArray(); - converts the queue to an array
boolean exists = queue.contains(element); - checks if element is in the queue
```

`Collections.min(numbers);`

`element = queue.clear();` - **Clearing the queue**

Сортиране – минаваме през друга структура от данни

Печатане / изход

Можем да итерираме елементите от Stack с foreach цикъл, но не можем да го обходим с нормален for цикъл. Т.е. нямаме достъп до индексите на опашката.

```
for (Integer elem : queue) {
    System.out.print(elem);
}
```

Priority Queue – в C# няма такава структура от данни

Ако няма зададен критерий, то ги сортира/подрежда по големина

```
public class PriorityQ {
    public static void main(String[] args) {
        PriorityQueue<Integer> queue = new PriorityQueue<>();
```

Същото като горния ред в случая е

```
PriorityQueue<Integer> queue =
new PriorityQueue<>(Comparator.comparingInt(Integer::intValue));
```

```

queue.offer(69);
queue.offer(13);
queue.offer(73);
queue.offer(42);

```

Без да има операции по опашката, имаме сортирана по Неар (купчина/пирамида) (hash таблицата). В случаите сортирани по MinHeap. Върхът на heap-а имплементиран чрез лист(зад която стои масив) е винаги нулевият елемент.

```

queue = [PriorityQueue@813] size = 4
    0 = {Integer@816} 13
    1 = {Integer@817} 42
    2 = {Integer@818} 73
    3 = {Integer@819} 69

```

Когато извършваме операции по опашката (добавяме/трием елемент), вече излизат подредени както следва: 13, 42, 69, 73 – пренареждат се както би трябвало да са след всяка операция

```

while (!numbers.isEmpty()) {
    System.out.println(numbers.poll());
}

```

В обратен ред

```

//      PriorityQueue<Integer> numbers = new
PriorityQueue<>(Comparator.comparingInt(Integer::intValue).reversed());

```

15. Многомерни масиви / Multidimensional Arrays

15.1. Деклариране

Стойности по подразбиране на елементите и при многомерните масиви е НУЛА.

ROWS	COLUMNS			
	[0][0]	[0][1]	[0][2]	[0][3]
	[1][0]	[1][1]	[1][2]	[1][3]
	[2][0]	[2][1]	[2][2]	[2][3]
	[3][0]	[3][1]	[3][2]	[3][3]

Row Index

Column Index

```

int[][][] intMatrix;
float[][][] floatMatrix;
String[][][] strCube;

```

`int[][][] intMatrix = new int[3][];` - поне дължината на редовете трябва да бъде декларирана при Java, а всеки ред от елементи може да се презаписва впоследствие

```

float[][][] floatMatrix = new float[8][2];
String[][][] stringCube = new String[5][5][5];

```

Initializing a multidimensional array with values:

```
int[][] matrix = {
    {1, 2, 3, 4}, // row 0 values
    {5, 6, 7, 8} // row 1 values
};
```

```
int[][] array = new int[][]{{1, 2}, {3, 4}};
```

Въвеждане на стойност с цикли - версия 1 – всеки елемент от масива е нов масив

```
int[][] arr = new int[rows][cols];
for (int r = 0; r < rows; r++) {
    arr[r] = Arrays.stream(sc.nextLine().split(" ")).  
        mapToInt(Integer::parseInt).toArray();
}
```

```
char[][] matrix = new char[r][c];
for (int i = 0; i < r; i++) {
    matrix[i] = sc.nextLine().toCharArray();
}
```

Въвеждане на стойност с цикли – версия 2

```
int[][] arr = new int[rows][cols];
for (int r = 0; r < rows; r++) {
    String[] elements = sc.nextLine().split(" ");
    for (int c = 0; c < elements.length; c++) {
        int number = Integer.parseInt(elements[c]);
        arr[r][c] = number;
    }
}
```

Въвеждане на стойност с цикли – версия 3:

```
int[][] matrix = new int[5][3]{};
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        matrix[i][j] = sc.nextInt();
    }
}
```

Въвеждане на стойност с цикли – от тип int – версия 4:

```
int[][] matrix = new int[rows][];
for (int i = 0; i < rows; i++) {
    matrix[i] = Arrays.stream(sc.nextLine().split(" "))
        .mapToInt(Integer::parseInt)
        .toArray();
}
```

от тип стринг

```
String[][] matrix = new String[rows][cols];
for (int i = 0; i < rows; i++) {
    matrix[i] = sc.nextLine().split("\s+");
}
```

Въвеждане на двумерен масив с метод:

```
int[][] matrix = readMatrix(rows,cols, sc);
```

```

private static int[][] readMatrix (int rows, int col, Scanner sc) {
    int[][] martix = new int[rows][col];
    for (int r = 0; r < rows; r++) {
        String[] elements = sc.nextLine().split(" ");
        for (int c = 0; c < elements.length; c++) {
            int number = Integer.parseInt(elements[c]);
            martix[r][c] = number;
        }
    }
    return martix;
}

```

Въвеждане на char

```

char[][] first = new char[rows][cols];
for (int row = 0; row < rows; row++) {
    String[] line = sc.nextLine().split("\\s+");
    for (int col = 0; col < cols; col++) {
        first[row][col] = line[col].charAt(0);
    }
}

```

Въвеждане на стойност

```

int[][] array = {{1, 2}, {3, 4, 5}}; array.length ще ни изведе 2 в случая
int element = array[1][1]; // element11 = 4

```

```

int[][] array = new int[3][4];
for (int row = 0; row < array.length; row++)
    for (int col = 0; col < array[0].length; col++)
        array[row][col] = row + col;

```

flatMap – прави от многомерни масиви на един поток/масив от елементи

```

public static int getElementsSumWithStream(int[][] matrix){
    return Arrays.stream(matrix)
        .flatMapToInt(arr -> Arrays.stream(arr))
        .sum();
}

```

Деклариране на матрица със списък(List)

```

List<List<Integer>> matrix = new ArrayList<>();
int counter = 1;
for (int i = 0; i < rows; i++) {
    List<Integer> numbers = new ArrayList<>();

    for (int j = 0; j < cols; j++) {
        numbers.add(counter++);
    }
    matrix.add(numbers);
}

```

Деклариране на матрица със списък(List)

```

private static void fillMatrix(List<List<Integer>> matrix, int rows, int cols) {
    int counter = 1;
    for (int row = 0; row < rows; row++) {
        matrix.add(new ArrayList<>());
        for (int j = 0; j < cols; j++) {
            matrix.get(row).add(counter++);
        }
    }
}

```

```
}
```

Когато трием от двумерен лист (лист от листи = матрица от листи), то е добре да започнем да трием от последния елемент от даден ред.

Границите на лист от листове е както следва:

```
private static boolean isInMatrix(int row, int col, List<List<Integer>> matrix) {  
    return row >= 0 && row < matrix.size() && col >= 0 && col < matrix.get(row).size();  
}
```

15.2. Операции

```
int[][] arr = new int[3][];  
System.out.println(arr.length); - връща броят масиви = брой редове от матрицата, т.е. връща 3  
System.out.println(arr[0].length); - връща елементите на всеки ред/масив = брой колони от реда на  
матрицата
```

Можем да обходим матрица с Foreach цикъл, но си губи смисъла – на кой ред и колона сме се губи като информация

Сумираме елементи на главния диагонал

```
for (int i = 0; i < matrix.length; i++) {  
    primarySum += matrix[i][i];  
}
```

Сума вторичен диагонал

```
int secondarySum = 0;  
for (int row = matrix.length - 1; row >= 0; row--) {  
    int col = matrix[0].length - 1 - row;  
    secondarySum += matrix[row][col];  
}
```

Когато работим с матрици, по-добре да използваме while цикли, като например:

```
//вторичен диагонал - вдясно и нагоре  
countRow = row;  
countCol = col;  
while (countRow >= 1 && countCol <= 6) {  
    countRow--;  
    countCol++;  
    if (matrix[countRow][countCol] == 'q') {  
        return false;  
    }  
}
```

15.3. Разни

Когато искаме да запазим координатите на матрица заедно със стойността на тази клетка.

```
List<int[]> updatedValues = new ArrayList<>(); //съхраняваме 3 стойности  
for (int row = 0; row < matrix.length; row++) {  
    for (int col = 0; col < matrix[row].length; col++) {  
        if (matrix[row][col] == wrongValue) {  
            updatedValues.add(new int[]{row, col, getClosestItemsSum(row, col, matrix, wrongValue)});  
        }  
    }  
}
```

Тук сменяме тези клетки от матрицата, които имат променени стойности

```

for (int[] updatedValue : updatedValues) {
    matrix[updatedValue[0]][updatedValue[1]] = updatedValue[2];
}

for (int i = 0; i < updatedValues.size(); i++) {
    matrix[updatedValues.get(i)[0]][updatedValues.get(i)[1]] = updatedValues.get(i)[2];
}

```

15.3. Печатане / изход

Печатане/обхождане с обикновен цикъл for

```

for (int row = 0; row < array.length; row++) {
    for (int col = 0; col < array[row].length; col++) {
        //array[row][col] = row + col;
        System.out.print(array[row][col]+ ",");
    }
    System.out.println();
}

```

Печатане

```

System.out.println(Arrays.toString(intMatrix[0]));

for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}

```

Печатане/обхождане с цикъл for (foreach ... iter)

```

int sum = 0;
int[][] matrix = readMatrix(sc, rows, cols, ", ");
for (int[] arr : matrix) {
    for (int num : arr) {
        sum+= num;
    }
}

```

Печатане на матрица от тип Лист

```

private static void printMatrix(List<List<Integer>> matrix) {
    for (int row = 0; row < matrix.size(); row++) {
        for (int col = 0; col < matrix.get(row).size(); col++) {
            System.out.print(matrix.get(row).get(col) + " ");
        }
        System.out.println();
    }
}

```

16. Sets and Maps Advanced

ВАЖНО: имаме ли Set, то трябва задължително да си имплементираме всеки път equals() и hashCode() методите – от Alt + Insert за по-лесно! Иначе, примерно ако създаваме random обекти, два различни обекта може да се окаже че имат еднакъв hashCode

```

import javax.persistence.*;
import java.util.Objects;

@Entity(name = "categories")

```

```

public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false, length = 15)
    private String name;

    public Category() {
    }

    public Category(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true; //сравнява по референция в паметта
        if (o == null || getClass() != o.getClass()) return false;
        Category category = (Category) o;
        return id == category.id; //сравняваме по уникалност само ключово поле id, защото само то
e уникално в SQL базата
        //дори да променим името на категорията, то тя все още е една и съща категория за базата
данни
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}

private Product sendRandomCategories(Product product) {
    Random random = new Random();
    long categoriesDbCount = this.categoryRepository.count();

    int count = random.nextInt((int) categoriesDbCount) + 1;

    Set<Category> categories = new HashSet<>();
    for (int i = 0; i < count; i++) {
        int randomId = random.nextInt((int) categoriesDbCount) + 1;

        Optional<Category> randomCategory = this.categoryRepository.findById(randomId);
        categories.add(randomCategory.get()); //при добавянето се бърка защото за Java си има
различен hashCode, но за базата hashCode трябва да го построим само по Id, а не по id и name.
        .get() връща от Optional<Category> само Category
    }

    product.setCategories(categories);
    return product;
}

List<Product> products = Arrays.stream(productsImportDTOS)
    .map(importDTO -> this.modelMapper.map(importDTO, Product.class))
    .map(this::setRandomSeller)
    .map(this::setRandomBuyer)
    .map(this::sendRandomCategories)
    .collect(Collectors.toList());
this.productRepository.saveAll(products);

```

16.1. Sets – това е Map, който пази само ключове

16.1.1. Описание

Пазят само уникални елементи

It offers very fast performance

`HashSet<E>` - Does not guarantee the constant order of elements over time

`TreeSet<E>` - The elements are ordered incrementally = ordered - **Uses a balanced search tree(BST)** – по-добре да не плащаме време за сортиранка накрая, а да плащаме по-малко време с използване на Tree

`LinkedHashSet<E>` - The order of appearance is preserved

16.1.2. Деклариране и въвеждане

```
Set<String> hash1 = new HashSet<String>();  
Set<Integer> hash2 = new HashSet<>();  
HashSet<String> swapped = new HashSet<>();
```

`HashSet<String> swapped = new HashSet<>(5); //с дължина 5 елемента`

`String[] inputCards = someInput.split(",\\s+");`
`Set<String> cards = new HashSet<>(Arrays.asList(inputCards));` - направи ми Сет от масива
inputCards като колекция

`Set<Character> separators = Set.of(',', '.', '!', '?');` - инициализираме по-различен Set колекция
от обикновената, с по-малко операции позволени

Или използваме

```
String[] inputCards = tokens[1].split(",\\s+");  
Set<String> cards; = new HashSet<>();  
Collections.addAll(cards, inputCards);
```

```
Set<String> tree = new TreeSet<>();  
Set<Double> linked = new LinkedHashSet<>();
```

Пример 1:

```
String[] input = sc.nextLine().split("\\s+");  
Set<String> strings = new HashSet<>();  
for (String str : input) {  
    strings.add(str);  
}
```

Пример 2:

```
String[] input = sc.nextLine().split("\\s+");  
Set<String> strings = new HashSet<>();  
Collections.addAll(strings, input);
```

Въвеждане с 1 ред от конзолата:

```
Set<Integer> firstPlayer = new LinkedHashSet<>();  
firstPlayer = Arrays.stream(sc.nextLine().split("\\s+"))  
    .mapToInt(Integer::parseInt)  
    .boxed()// за видигане на числа  
    .collect(Collectors.toCollection(LinkedHashSet::new));
```

За сваляне на типа – от Double към double

```
.forEach(x -> {
    Double average = Arrays.stream(x.getValue()) Stream<Double>
        .mapToDouble(Double::doubleValue) DoubleStream
        .average() OptionalDouble
        .orElse( other: 0);
```

За сваляне на типа – от Integer към int

```
int[] numbers = Arrays.stream(sc.nextLine().split(", "))
    .mapToInt(num -> Integer.parseInt(num))
    .boxed()
    .mapToInt(Integer::intValue) или .mapToInt(e -> e)
    .filter(num -> num % 2 == 0)
    .toArray();
```

1) Обхождане на колекция с нормален for цикъл плюс Iterator

```
words = Arrays.stream(sc.nextLine().split(", ")).collect(Collectors.toList());
words.removeIf(next -> !target.contains(next));
```

```
for (Iterator<String> iter = words.iterator(); iter.hasNext(); ) { //Взема първият елемент от
колекцията, след това проверява дали има следващ.
    String next = iter.next(); //ако има следващ, то влизаме в тялото и посочваме следващият елем.
    if (!target.contains(next)) {
        iter.remove();
    }
}
```

2) от stream в каквато и да е колекция

```
LinkedHashSet<Integer> secondPlayer = Arrays.stream(sc.nextLine().split("\s+"))
    .mapToInt(Integer::parseInt)
    .boxed()// за вдигане на тупа
    .collect(Collectors.toCollection(LinkedHashSet::new));
int secondCard = secondPlayer.iterator().next();
```

```
Set<Integer> firstPlayer = new LinkedHashSet<>();
firstPlayer = Arrays.stream(sc.nextLine().split("\s+"))
    .mapToInt(Integer::parseInt)
    .boxed()// за вдигане на тупа
    .collect(Collectors.toCollection(LinkedHashSet::new));
```

3) Още малко за iterator

```
Iterator<Integer> firstIterator = firstPlayer.iterator();
int firstCard = firstIterator.next();
firstIterator.remove(); == firstPlayer.remove(firstCard);
int firstNumber = firstPlayerCards.iterator().next();
firstPlayerCards.remove(firstNumber);
```

4) От лист към сет

```
List<String> list = Arrays.stream(sc.nextLine().split("\s+"))
    .collect(Collectors.toCollection(ArrayList::new));
System.out.println(String.join(" ", list));

LinkedHashSet<String> set = new LinkedHashSet<>(list);
System.out.println(String.join(" ", set));
```

Обратното също важи – от сет към лист

```
LinkedHashSet<String> set = Arrays.stream(sc.nextLine().split("\\s+"))
    .collect(Collectors.toCollection(LinkedHashSet::new));
System.out.println(String.join(" ", set));

List<String> list = new ArrayList<>(set);
System.out.println(String.join(" ", list));
```

16.1.3. Методи/операции

Нямаме индексация реално. Можем само да итерираме, но нямаме достъп по конкретен индекс. **Нямаме .get()**

```
tree.size();
hash1.isEmpty(); което е hash1.size() == 0;
el.hashCode(); - връща числа за всеки един елемент (обикновено от ASCII таблицата + доп.)
strings.toArray(); - не винаги ще се случи, това което очакваме да се случи
```

```
Set<Integer> numbers = new HashSet<>();
for (int i = 0; i < 10; i++) {
    numbers.add(i);
}

public class Test {
    public static void main(String[] args) {
        Set<Integer> numbers = new HashSet<>();
        boolean isAdded = numbers.add(1);
        System.out.println(isAdded); - връща true

        isAdded = numbers.add(1);
        System.out.println(isAdded); - връща false
    }
}
```

Добавяне на колекция към края на сета

```
firstPlayer.addAll(Arrays.asList(firstCard, secondCard));
```

carNumbers.remove(registration); - remove object

```
Set<String> arrivedGuests = new LinkedHashSet<>();
vip.removeAll(arrivedGuests); - премахва всички vip-ове, които се съдържат в колекцията
arrivedGuests
```

```
Set<String> vip = new TreeSet<>();
removeIf - да избягваме да го ползваме - цикли всички елементи и не проверява само по ключ
```

aaa.clear(); - премахва всички елементи на структурата

```
Set<String> first = new HashSet<>();
Set<String> second = new HashSet<>();
```

```
first.add("First");second.add("First");
first.add("Second");second.add("Second");
first.add("Third");
```

```
first.retainAll(second); // намира сечението на first и second, като first става самото сечение
връща boolean
```

```
for (String s : first) {  
    System.out.println(s);  
}
```

Обратно сортиране при използване на TreeSet структурата от данни (ако не искаме да сортираме накрая):

```
Set<Integer> numbers = new TreeSet<>((f,s) -> Integer.compare(s, f));
```

Или

```
Comparator<Integer> comparator = (f,s) -> Integer.compare(s, f);  
Set<Integer> numbers = new TreeSet<>(comparator);
```

16.1.4. Печатане / изход

```
Set<String> strings = new HashSet<>();  
for (String el : strings) {  
    System.out.println(el);  
}  
  
userNames.stream()  
    .forEach(x -> System.out.println(x));
```

16.2. Maps = Associative Arrays



Stream debugging / Дебъгване на stream

Trace current stream chain

Как обхождаме вложени мапове без Stream API – ВАЖНО:

```
Map<String, Map<String, List<String>>> allData = new LinkedHashMap<>();  
for (Map.Entry<String, Map<String, List<String>>> entry : allData.entrySet()) {  
    System.out.println(entry.getKey() + ":" + entry.getValue());  
  
    for (Map.Entry<String, List<String>> innerEntry : entry.getValue().entrySet()) {  
        System.out.println(" " + innerEntry.getKey() + " -> " + String.join(", ",  
innerEntry.getValue()));  
    }  
}
```

Когато имаме Map с елемент Set като част от Map-а:

```
Map<String, LinkedHashSet<String>> players = new LinkedHashMap<>();  
players.putIfAbsent(name, new LinkedHashSet<>());  
LinkedHashSet<String> strings = players.get(name);  
String[] hand = tokens[1].split(",\\s+");  
strings.addAll(Arrays.asList(hand)); - добавяме нови елементи в сета  
players.put(name, strings); - обновяваме новия сет в мапа
```

Анонимно сътвржане/добавяне на елементи (има и всички други команди на съответната структура данни)

```
TreeMap<String, LinkedHashMap<String, Integer>> usersLogs = new TreeMap<>();
```

```
usersLogs.put(username, new LinkedHashMap<>(){put(ip, 1)}); - достъпваме в случая вътрешния  
LinkedHashMap анонимно.
```

Горното е същото като този запис:

```
usersLogs.put(username, new LinkedHashMap<>());
usersLogs.get(username).put(ip, 1);
```

Когато работим с две отделни структури от данни с еднакви ключове

```
TreeMap<String, Integer> durations = new TreeMap<>();
HashMap<String, TreeSet<String>> ips = new HashMap<>();
for (Map.Entry<String, Integer> entry : durations.entrySet()) {
    String userName = entry.getKey();
    System.out.printf("%s: %d [%s]\n", userName, entry.getValue(),
                      String.join(", ", ips.get(userName)));
}
```

Въвеждане на Map с помощта на IntStream

```
int numberOfStudents = Integer.parseInt(sc.nextLine());
TreeMap<String, Double> graduationList = new TreeMap<>();

public static IntStream range(int startInclusive, int endExclusive) {}
IntStream.range(0, numberOfStudents) - без последния
    .mapToObj(i -> sc.nextLine())
    .forEach(name -> graduationList
        .put(name,
              Arrays.stream(sc.nextLine().split("\s+"))
                    .map(Double::parseDouble).collect(Collectors.toList())))
    )
);
```

Въвеждане на Map с помощта на IntStream

```
String oddOrEven = sc.nextLine();
Predicate<Integer> filter = getFilter(oddOrEven);
Consumer<Integer> printer = x -> System.out.print(x + " ");

public static IntStream rangeClosed(int startInclusive, int endInclusive) {}
IntStream.rangeClosed(lower, upper) - включва от първия до последния включително
    .boxed()
    .filter(filter)
    .forEach(printer);
```

Възможност за създаване на МАР чрез `Collectors.toMap` – но може да има конфликти при дублиране на елементи

```
String str = "Hello, hello, helo, heo";
Map<String, Integer> mapCollect = Arrays.stream(str.split(", "))
    .collect(Collectors.toMap(e -> e, e -> e.length()));
```

Въвеждане на map от конзолата – как го четем/записваме? Ето:

```
public class TestsMaps {
    public static class Person {
        String name;
        int age;

        public Person(String name, int age) {
            this.name = name;
            this.age = age;
        }
    }

    public static void main(String[] args) {
```

```

Scanner sc = new Scanner(System.in);
//Pesho 12, Gosho 13, Ivan 42
Map<String, Person> strings = Arrays.stream(sc.nextLine().split(","))
    .map(str -> {
        String[] tokens = str.split("\s+");
        return new Person(tokens[0], Integer.parseInt(tokens[1]));
    })
    .collect(Collectors.toMap(p->p.name, p -> p));

for (Map.Entry<String, Person> entry : strings.entrySet()) {
    System.out.println(entry.getKey() + " " + entry.getValue().name + " " +
entry.getValue().age);
}
}

```

Как се декларира двойка Pair или key-value pair:

Map.Entry<String, String>

Лист от единични двойки - pairs – така може да създадем повтарящи се ключове

Вариант 1 – елегантен начин

```

List<Map.Entry<String, Integer>> test = new LinkedList<>();
test.add(Map.entry("a", 2));
test.add(Map.entry("b", 3));
test.add(Map.entry("a", 4));

```

```

List<Map.Entry<String, double[]>> studentGrades = new ArrayList<>();
while (n-- > 0) {
    String name = sc.nextLine();
    double[] grades = Arrays.stream(sc.nextLine().split("\s+"))
        .mapToDouble(Double::parseDouble)
        .toArray();

    studentGrades.add(Map.entry(name, grades));
}

studentGrades.stream()
    .sorted((x1, x2) -> {
        String name1 = x1.getKey();
        String name2 = x2.getKey();
        return name1.compareTo(name2);
    })
    .forEach(x -> {
        double average = Arrays.stream(x.getValue()).average().orElse(0);
        String name = x.getKey();
        System.out.println(String.format("%s is graduated with %s", name,
            new DecimalFormat("0.#####").format(average)));
    });
}

```

Как да запазим данни от API stream от Map и да го преобразуваме в Map.Entry и да го използваме след това

LinkedHashMap<String, Integer> countriesPopulation = new LinkedHashMap<>();

```

List<Map.Entry<String, Integer>> orderedCountriesPopulation = countriesPopulation.entrySet().stream()
    .sorted((f, s) -> {
        return s.getValue().compareTo(f.getValue());
    });

```

```

})
.collect(Collectors.toList());

for (Map.Entry<String, Integer> entry : orderedCountriesPopulation) {
    String country = entry.getKey();
    System.out.printf("%s (total population: %d)%n", country, entry.getValue());
    LinkedHashMap<String, Integer> innerEntry = countriesCitiesPopulation.get(country);
    innerEntry.entrySet().stream()
        .sorted((f, s) -> {
            return Integer.compare(s.getValue(), f.getValue());
        })
        .forEach(x-> {
            System.out.printf("=>%s: %d%n", x.getKey(), x.getValue());
        });
}
}

```

Вариант 1 – сложен начин

```

int n = Integer.parseInt(sc.nextLine());
List<Map<String, double[]>> studentGrades = new ArrayList<>();
while (n-- > 0) {
    String name = sc.nextLine();
    double[] grades = Arrays.stream(sc.nextLine().split("\s+"))
        .mapToDouble(Double::parseDouble)
        .toArray();

    HashMap<String, double[]> student = new HashMap<>();
    student.put(name, grades);

    studentGrades.add(student);
}

studentGrades.stream()
    .sorted((x1, x2) -> {
        String name1 = null;
        for (Map.Entry<String, double[]> entry : x1.entrySet()) {
            name1 = entry.getKey();
        }

        String name2 = null;
        for (Map.Entry<String, double[]> entry : x2.entrySet()) {
            name2 = entry.getKey();
        }

        return name1.compareTo(name2);
    })
    .forEach(x -> {
        double average = 0;
        for (Map.Entry<String, double[]> entry : x.entrySet()) {
            average = Arrays.stream(entry.getValue()).average().orElse(0);
        }

        String name = null;
        for (Map.Entry<String, double[]> entry : x.entrySet()) {
            name = entry.getKey();
        }

        System.out.printf("%s is graduated with %s%n", name, average);
    });
}

```

Вариант 2:

The `Pair` class can be found in the `javafx.util` package

```
Pair<Integer, String> pair = new Pair<>(1, "One");
Integer key = pair.getKey();
String value = pair.getValue();
```

Вариант 3:

Class `MainPair` – ние си го създаваме

```
List<ManualPair> pairs = new ArrayList<>();

public class ManualPair {
    private String keyParent;
    private String valueChild;

    public ManualPair(String keyName, String valueBirthday) {
        this.keyParent = keyName;
        this.valueChild = valueBirthday;
    }

    public String getKeyParent() {
        return keyParent;
    }

    public String getValueChild() {
        return valueChild;
    }

    public void setKeyParent(String keyParent) {
        this.keyParent = keyParent;
    }

    public void setValueChild(String valueChild) {
        this.valueChild = valueChild;
    }
}
```

getOrDefault on Map

```
Map<Integer, Integer> mapNumberFrequency = new HashMap<>();
int valueMapNumberFrequency_OLD = mapNumberFrequency.getOrDefault(numToAddRemoveCheck, 0); //нула е дефолтната стойност
```

```
Map<Integer, Set<Integer>> mapFrequencyNumber = new HashMap<>();
if (mapFrequencyNumber.getOrDefault(numToAddRemoveCheck, Collections.emptySet()).size() > 0)
//празен сет е новата стойност
```

17. Streams, Files and Directories

1. Streams Basics - Streams are used to transfer data (from files)

Stream API is different!

It is a **declarative programming** paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

1. Read File

Вариант 1 - за четене на файл – по редове или по думи – чрез Scanner и FileInputStream

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner sc = new Scanner(new FileInputStream("C:\\\\Users\\\\svilk\\\\Desktop\\\\Hello.txt")); или
    Scanner sc = new Scanner(new FileInputStream("C:/Users/svilk/Desktop/Hello.txt"));

    String input = sc.nextLine(); - прочети първия ред
    String input1 = sc.next(); - прочети първата дума от следващия ред

    System.out.println(input); - отпечатва първия ред
    System.out.println(input1); - отпечатва една дума от втория ред

    while (sc.hasNext()) { - кога стигаме края на файла при използване на Scanner
        lineNumber++;
        if (lineNumber % 3 == 0) {
            System.out.println(sc.nextLine()); - отпечатва текущия ред и минава на следващ ред
        } else {
            sc.nextLine(); - отиди на следващия ред
        }
    }
}
```

Текуща директория

Scanner sc = new Scanner(new FileInputStream("Hello.txt")); - файлът Hello.txt трябва да сме го създали предварително ръчно

Name	Status
.idea	✓
out	✓
src	✓
04. Java-Advanced-Streams-Files-and-Director...	✓
04. Java-Advanced-Streams-Files-and-Director...	✓
04. Java-Advanced-Streams-Files-and-Director...	✓
09 - Streams, files and directories - LAB.iml	✓
Hello.txt	✓

Вариант 2 – за четене на файл - по цели редове – чрез класа Files и метода readAllLines = BufferedReader

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

```
String filename = "Input.txt"
Path path = Paths.get(filename);
Path path = Path.of(filename);
```

```
List<String> lines = Files.readAllLines(path); //реално използва BuffRead.
for (String string : strings) {
    System.out.println(string);
}
```

Вариант 3 - за четене от файл - по букви без space-вете и новите редове – FileInputStream без да използваме Scanner

```
FileInputStream fileStream = new FileInputStream("Hello.txt");
int oneByte = fileStream.read(); - чете следващия byte/буква/символ
while (oneByte >= 0) {
    System.out.print((char)oneByte);
    oneByte = fileStream.read(); - пропуска да прочете space/нов ред
}
```

Вариант 4 – за четене на файл по байтове

```
FileInputStream inputStream = new FileInputStream(path);
int nextByte = inputStream.read();

while (nextByte != -1) {
    System.out.print(Integer.toBinaryString(nextByte) + " ");
    nextByte = inputStream.read();
}
```

Вариант 5 – четене с FileReader

```
FileReader reader = new FileReader(path); - използва реално FileInputStream
Scanner scanner = new Scanner(reader);

while (scanner.hasNext()) { - има ли следваща дума
    if (scanner.hasNextInt()) {
        System.out.println(scanner.nextInt());
    }
    scanner.next(); - взема следващата дума
}
```

Вариант 6 – с използване на класа Files и метода newBufferedWriter

```
BufferedReader reader = Files.newBufferedReader(path);
String line = reader.readLine();
while (line != null) {
    int sum = 0;
    for (char symbol : line.toCharArray()) {
        sum += symbol;
    }
    System.out.println(sum);

    line = reader.readLine();
}
```

Отваряне и затваряне на входящ поток

```
FileInputStream inputStream = new FileInputStream("Hello.txt"); - отваряне на потока
inputStream.available();
inputStream.close(); - затваря се потока, след което нямаме достъп до него
inputStream.read();
inputStream.readLine();
```

Try-catch-finally and FileInputStream

Try-catch е бавно-работеща конструкция. Ако можем проверка с if-else, то нямаме право да използваме try-catch.

1) Closing a File Stream Using try-catch-finally block

```

String output;
FileInputStream inputStream = null;
try {
    inputStream = new FileInputStream("Hello.txt");
    output = "File found";
    return; // - първо изпълнява finally, и след това затваря main метода
} catch (FileNotFoundException ex){
    output = "File not found";
} finally { //Винаги ще се изпълни
    inputStream.close(); //освобождава памет/място
}

```

2) Closing a File Stream Using try-with-resources block (try with resources)

```

String path = "Hello.txt";
try (InputStream in = new FileInputStream(path)) {
    int oneByte = in.read();
    while (oneByte >= 0) {
        System.out.print(oneByte);
        oneByte = in.read();
    }
} catch (IOException e) {
    TODO: handle exception
} finally { //Винаги ще се изпълни
    inputStream.close(); //освобождава памет/място
}

```

2. Write to a file

Когато пишем в даден файл, той сам се създава, например new FileOutputStream("output.txt"); създава файла output.txt

Вариант 1 - Записване на данни във файл – FileOutputStream и PrintWriter

```

public static void main(String[] args) throws IOException {
    String path = "C:\\\\Users\\\\svilk\\\\OneDrive\\\\Soft Engineer\\\\JAVA\\\\Advanced\\\\prepare - May 2020\\\\09 - Streams, files and directories - LAB\\\\input.txt";

    File file = new File(path);
    FileInputStream inputStream = new FileInputStream(file);
    Scanner sc = new Scanner(inputStream);

    StringBuilder builder = new StringBuilder();
    String line = sc.nextLine();

    while (line != null) {
        builder.append(line.replaceAll("[,.!?]", "")).append(System.lineSeparator());
        try {
            line = sc.nextLine();
        } catch (NoSuchElementException ex) {
            inputStream.close();
            break;
        }
    }

    String string = builder.toString();

    FileOutputStream outputStream = new FileOutputStream("output.txt"); //локално се създава файла в проекта
    PrintWriter printWriter = new PrintWriter(outputStream);
    printWriter.append(string);

    printWriter.print(string);
    printWriter.flush(); // дай му запис във файла. Ако му дадем .close, то няма да можем да запишем тези
    // данни и в друг файл
}

```

Вариант 2 – веднага записваме във файла – с използване на Writer и FileWriter

```
public static void main(String[] args) throws IOException {
    String path = "C:\\\\Users\\\\svilk\\\\OneDrive\\\\Soft Engineer\\\\JAVA\\\\Advanced\\\\prepare - May
2020\\\\09 - Streams, files and directories - LAB\\\\input.txt";
    File file = new File(path);
    byte[] bytes = Files.readAllBytes(file.toPath());

    Writer writer = new FileWriter("text-as-bytes.txt"); //директно пише във файла
    for (byte b : bytes) {
        String out = String.valueOf(b);
        if (b == 32) {
            out = " ";
        } else if (b == 10) {
            out = System.lineSeparator();
        }
        writer.write(out); //директно пише във файла
    }
    writer.flush(); // последната част да се нанесе във файла с тази команда
}

FileWriter fileWriter = new FileWriter("out.txt");
fileWriter.write(asciiSum + "\n");
fileWriter.flush(); - //предай данните по-нататък, и изчисти обекта от текущите данни
fileWriter.close();
```

Вариант 3 – с използване на BufferedWriter

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

List<String> lines = new ArrayList<>();
String line = reader.readLine();
while (line != null){
    lines.add(line);
    line = reader.readLine();
}

Collections.sort(lines);

BufferedWriter writer = new BufferedWriter(new FileWriter("out-test.txt"));
writer.write(String.join(System.lineSeparator(), lines));
writer.flush(); //последната част да се нанесе във файла с тази команда
writer.close();

-----
FileWriter fileWriter = new FileWriter("new file.txt");
BufferedWriter writer = new BufferedWriter(fileWriter);
```

Вариант 4 – с използване на PrintStream

```
FileInputStream inputStream = new FileInputStream(path);

int nextByte = inputStream.read();
Set<Character> separators = Set.of(',', '.', '!', '?');
PrintStream printStream = new PrintStream("out.txt"); //локално се създава файла в проекта

while (nextByte != -1){
    char symbol = (char) nextByte;
```

```

if (!separators.contains(symbol)) {
    printStream.print(symbol);
}

nextByte = inputStream.read();
}

```

Вариант 5 – с използване на класа **Files** и метода **write**

```

List<String> lines = new ArrayList<>();
String line = reader.readLine();
while (line != null){
    lines.add(line);
    line = reader.readLine();
}
Files.write(Path.of("sorted-lines.txt"), lines); - създай файл и копирай всички
редове в този файл

```

Вариант 6 – с използване на класа **Files** и метода **newBufferedWriter**

```

BufferedWriter writer = Files.newBufferedWriter(Path.of("test-out.txt"));
String line = reader.readLine();
while (line != null) {
    int sum = 0;
    for (char symbol : line.toCharArray()) {
        sum += symbol;
    }
    System.out.println(sum);
    writer.write(sum); // сериализира го тук

    line = reader.readLine();
}
writer.flush();
writer.close();

```

2. Types of Streams

Byte stream

Byte streams are the **lowest level streams**

Byte streams can read or write **one byte at a time**

All byte streams **descend** from **InputStream** and **OutputStream** – с други думи тези 2 потока са за byte-ве.

Character stream

All character streams descend from **FileReader** and **FileWriter** – с други думи тези 2 потока са за Character

```

String path = "D:\\input.txt";
FileReader reader = new FileReader(path);
FileWriter fileWriter = new FileWriter("out.txt");

```

Character streams are often "wrappers" for byte streams

- **FileReader** uses **FileInputStream**
- **FileWriter** uses **FileOutputStream**

Wrapping a Stream

```
String path = "D:\\input.txt";  
Scanner reader = |  
    new Scanner(new FileInputStream(path));
```

```
Path path = Path.of("src/main/resources/output/outputToJSON.json");  
FileWriter fileWriter = new FileWriter(String.valueOf(path));  
fileWriter.write(content);  
fileWriter.close();
```

Вземане само на числа от даден файл

```
public static void main(String[] args) throws FileNotFoundException {  
    File file = new File("input.txt");  
    Scanner sc = new Scanner(file);  
    PrintWriter writer = new PrintWriter("integer.csv");  
  
    while (sc.hasNext()) {  
        if (sc.hasNextInt()) {  
            int nextInt = sc.nextInt();  
            writer.println(nextInt);  
        }  
        sc.next();  
    }  
    writer.flush();  
}
```

Buffered Streams – от конзола или от файл

Reading information in **chunks**

Significantly **boost performance**

```
File file = new File("input.txt");  
FileInputStream inputStream = new FileInputStream(path);  
BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));  
Или  
BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(path)));  
- от външен файл
```

Или

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in)); - от конзолата
```

Чрез **BufferedReader** от конзолата четем:

```
public static void main(String[] args) throws IOException {  
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));  
    String hello = reader.readLine(); // Hello BufferedReader  
    List<String> listLines = reader.lines().collect(Collectors.toList()); // Hello BufferedReader  
    System.out.println(hello); // Hello BufferedReader  
}
```

String input = reader.readLine(); - чете цял ред и отива на следващия
String input = reader.read(); - чете една дума от текущия ред

Чрез BufferedReader от файл четем:

```
FileInputStream inputStream = new FileInputStream(path);
BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
String line = reader.readLine(); - вземи първия ред от файла
while (line != null){ -
    boolean
    line = reader.readLine();
}
```

Command Line I/O – само от конзолата

Стандартни:

Standard Input - **System.in**
Standard Output - **System.out**
Standard Error - **System.err**

```
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine();
System.out.println(line);
System.err.println("Error");
```

3. Files and Directories

Директорията е файл, който в себе си съдържа други файлове

Files and Paths

```
String fileName = "input.txt";
String dir = System.getProperty("user.dir");
System.out.println(dir + "\\\" + fileName);
```

Path path = Paths.get("D:\\\\input.txt"); - Represented in Java by the *Paths* class
Path path = Path.of("sorted-lines.txt") - Represented in Java by the *Path* class

Provides static methods for creating streams

```
Path path = Paths.get("D:\\\\input.txt");
try (BufferedReader reader =
      Files.newBufferedReader(path)) {
    // TODO: work with file
} catch (IOException e) {
    // TODO: handle exception
}
```

Provides utility methods for easy file manipulation

```
Path inPath = Paths.get("D:\\\\input.txt");
Path outPath = Paths.get("D:\\\\output.txt");
List<String> lines = Files.readAllLines(inPath);
Files.write(outPath, lines);
```

File Class in Java

```
File file = new File("new-file.txt"); //create new file in Java, not in the operating system
File directory = new File("newDir"); //create new folder in Java, not in the operating system
File file = new File("D:\\\\input.txt"); //създава файл input.txt in Java, not in the operating system
```

```

File file = new File("out.txt");
file.delete(); - изтрива файла
file.createNewFile(); - създай файл с името out.txt

boolean exists = file.exists(); - дали съществува файла
long length = file.length(); // дава/връща размерът в байтове bytes
boolean isDirectory = file.isDirectory(); //дали файлът е директория
file.isFile(); //дали файлът е файл

File[] files = file.listFiles(); //Показва всички файлове и папки в дадената папка (1 ниво надолу)
String[] arrStrings = file.list(); //Показва всички файлове и папки в дадената папка (1 ниво
надолу) под формата на String

file.getName() - връща името на файла/директорията
file.canExecute(); - дали файлът е executable
file.canRead(); - дали може да се чете от файла
file.canWrite(); - дали е read-only
file.compareTo(file2); - дали един файл е същия като друг файл

```

Nested folders – с Breadth-First Search (BFS – на вълни = на широчина) и без рекурсия – колко папки имаме

```

File file = new File("Files-and-Streams");
Deque<File> queue = new ArrayDeque<>();
queue.offer(file);
int count = 0;

while (!queue.isEmpty()) {
    File current = queue.poll();
    File[] nestedFiles = current.listFiles();

    for (File f : nestedFiles) {
        if (f.isDirectory()) {
            queue.offer(f);
        }
    }

    count++;
    System.out.println(current.getName());
}
System.out.println(count + " folders");

```

Обхождане на директории и принтиране на всички файлове – с рекурсия

```

public class PrintFiles_Directories {
    public static void main(String[] args) {
        File folder = new File("D:\\Video\\Figuri_tanci_uroci\\2014");

        printAllFilesInAllFolders(folder);
    }

    private static void printAllFilesInAllFolders(File folder) {
        File[] listOfCurrentDirectories = folder.listFiles(file -> file.isDirectory());
        File[] listOfCurrentFilesToPrint = folder.listFiles(file -> !file.isDirectory());

        for (File currentFileToPrint : listOfCurrentFilesToPrint) {
            System.out.println(currentFileToPrint.getName());
        }

        for (File currentDirectory : listOfCurrentDirectories) {

```

```
        printAllFilesInAllFolders(currentDirectory);
    }
}
```

4. Serialization – обмен на обекти и записването им под друг начин

4.1. *Serialization* - Save objects to a file – изпраща/кодира все едно

```
public class SerializeCustomObject_09 {
    public static class Cube implements Serializable { - прави всяка инстанция на класа Cube да
може да се сериализира
        private String name;
        private int width;
        private int length;
        private int height;

        public Cube(String name, int width, int length, int height) {
            this.name = name;
            this.width = width;
            this.length = length;
            this.height = height;
        }
    }

    public static void main(String[] args) throws IOException {
        Cube cube = new Cube("Ice Cube", 13, 42, 69);

        //Сериализация
        FileOutputStream fos = new FileOutputStream("obj.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(cube);
        oos.close();

        // Десериализация в рамките на същото изпълнение на програмата
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.txt"));
        Cube savedCube = (Cube) ois.readObject();
    }
}
```

Преобразуване на един тип данни в stream bytes

```
List<Integer> list = List.of(13, 42, 43, 98, 73);
FileOutputStream fos = new FileOutputStream("ser.txt");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(list);
oos.close();
```

Сериализация на лист от Integer

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
ObjectOutputStream objectOutputStream =
    new ObjectOutputStream(new FileOutputStream("src/resources/list.ser"));
for (Integer number : numbers) {
    objectOutputStream.write(number);
}
```

How to copy a picture from one file to another

```
FileInputStream fis = new FileInputStream(new File("src\\QVNIMTE0MzQ0MzA4.jpg"));
FileOutputStream outputStream = new FileOutputStream(new File("src\\destination.jpg"));
byte[] buffer = new byte[1024];
```

```

int read = 0;
while ((read = fis.read(buffer)) > 0) {
    outputStream.write(buffer, 0, read);
}

```

4.2. Deserialization - Load objects from a file – разчита/разкодира все едно

Преобразуване от stream bytes в друг тип данни

```

FileInputStream fis = new FileInputStream("ser.txt");
FileInputStream fis = new FileInputStream(new File("src\\QVNIMTE0MzQ0MzA4.jpg"));

ObjectInputStream ois = new ObjectInputStream(fis);
List<Integer> result = (List<Integer>)ois.readObject(); //кастваме към Лист от Integer

for (Integer r : result) {
    System.out.println(r);
}

```

4.3. Custom objects should implement the Serializable interface

```

class Cube implements Serializable {
    String color;
    double width;
    double height;
    double depth;
}

String path = "D:\\save.ser";
Cube cube = new Cube();
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(path))) {
    oos.writeObject(cube);
} catch (IOException e) {
    e.printStackTrace();
}

```

5. How to zip a file

```

public static void main(String[] args) throws IOException {
    ZipOutputStream zos = new ZipOutputStream(new FileOutputStream(new
File("src/resources/textfiles.zip")));
    FileInputStream fis = new FileInputStream(new File("src/resources/words.txt"));
    int byteContainter;

    zos.putNextEntry(new ZipEntry("words.txt"));
    while ((byteContainter = fis.read()) != -1) {
        zos.write(byteContainter);
    }
    zos.closeEntry();

    zos.putNextEntry(new ZipEntry("text.txt"));
    fis = new FileInputStream(new File("src/resources/text.txt"));
    while ((byteContainter = fis.read()) != -1) {
        zos.write(byteContainter);
    }
    zos.closeEntry();

    zos.putNextEntry(new ZipEntry("input.txt"));
    fis = new FileInputStream(
        new File("src/resources/input.txt"));
    while ((byteContainter = fis.read()) != -1) {
        zos.write(byteContainter);
    }
}

```

```

    zos.closeEntry();

    zos.finish();
    zos.close();
}

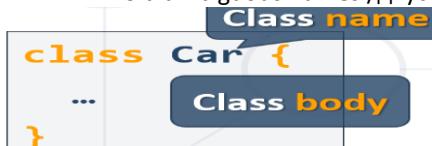
```

18. Defining Classes

1. Defining Classes

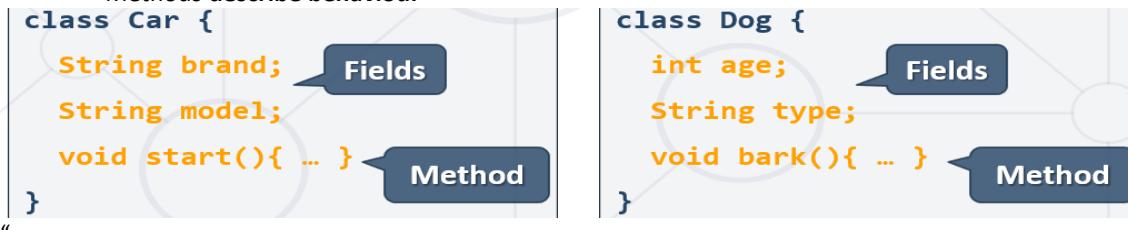
Naming Classes

- Use PascalCase naming
- Use descriptive nouns
- Avoid ambiguous names /двуислен/

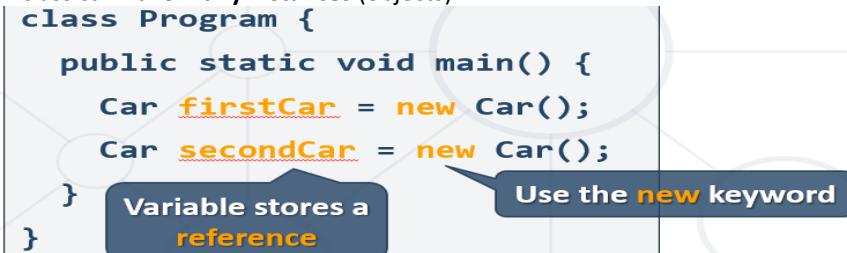


Class is made up of **state** and **behavior**:

- Fields **store state**
- Methods **describe behaviour**



A class can have **many instances** (objects)



Object Reference

- Declaring a variable creates a **reference** in the **stack** – **автоматична** променлива - In computer programming, an automatic variable is a local variable which is allocated and deallocated automatically when program flow enters and leaves the variable's scope. Automatic local variables **are normally allocated in the stack frame** of the procedure in which they are declared.[
- The **new keyword** allocates memory on the **heap**



Classes vs. Objects

Classes provide structure for creating objects

An object is a single instance of a class

2. Class Data

Fields

Class fields have **access modifiers**, **type** and **name**

access modifier

```
public class Car { type  
    private String brand; name  
    private int year;  
    public Person owner;  
    ...  
}
```

Fields can be of any type

Access Modifiers

- Classes and class members **have modifiers**
- Modifiers **define visibility**

Class modifier

```
public class Car {  
    private String brand;  
    private String model;  
}
```

Member modifier

Fields should always be private!

Methods

Store **executable code** (algorithm) that manipulate state

```
public class Car {  
    private int horsePower;  
  
    public void increaseHP(int value) {  
        horsePower += value;  
    }  
}
```

Getters and Setters

```
class Car {  
    Field is hidden  
    private int horsePower;  
    public int getHorsePower() {  
        return this.horsePower;  
    }  
    this points to the current instance  
    public void setHorsePower(int horsePower) {  
        this.horsePower = horsePower;  
    }  
}
```

Getter provides access to field

Setter provide field change

Keyword this

- Prevent field hiding
- Refers to a current object

Ако не използваме **this** в сътъра **setHorsePower**, тогава **horsePower** винаги ще е 0.

Десен бутон + Generate или Alt+Insert

ToString() Method - Whenever we try to print the Object reference, then internally `toString()` method is invoked.

```
Car car = new Car();
System.out.println(car); //Car@3feba861
```

If you define `toString()` method in your class then your implemented/Overridden `toString()` method will be called..

Alt+Insert -> Override methods -> `toString()`

Може да използваме и без `@Override`, но ако цитираме нещо грешно, то `@Override` няма да може да ни предпази/сигнализира за грешка

```
@Override
public String toString(){
    return getBrand() + " " + getModel() + " " + getHorsePower();
}
```

Equals() Method

```
boolean isEqual = firstCar.equals(secondCar);
```

HashCode() Method

```
Car car = new Car();
car.setBrand("Chevrolet");
car.setModel("Impala");
car.setHorsePower(390);
int hash = car.hashCode(); //integer value which represents hashCode value for this class.
System.out.println(hash);
```

Constructors

The only one way to **call a constructor** in Java is through the **keyword new**

Special methods, executed during object creation

Default constructor:

```
public Car(){
}
```

Overload-ване на дефолтния конструктор:

1) Constructor with zero parameters и с hardcore-нати дефолтни стойности – винаги ще бъде BMW

```
public Car() {
    this.brand = "BMW";
}
```

2) Constructor with all parameters

```
public Car(String brand, String model, int horsePower) {
    this.brand = brand;
    this.model = model;
    this.horsePower = horsePower;
}
```

Constructors set object's initial state

```
public class Car {
    String brand;
    List<Part> parts;

    public Car(String brand) {
```

```

        this.brand = brand;
        this.parts = new ArrayList<>();
    }
}

```

Constructor Chaining - Constructors can call each other – да внимаваме да не създадем рекурсия!!!

Идеята е конструкторът с по-малко параметри да извиква конструкторът с повече параметри!!!

```

public class Car {
    private String brand;
    private String model;
    private int horsePower;

    public Car(String brand) {
        this(brand, "unknown", -1); // извикване на конструкторът с 3 параметъра
        // this.brand = brand;
        // this.model = "unknown";
        // this.horsePower = -1;
    }

    public Car(String brand, String model) {
        this(brand, model, -1); // извикване на конструкторът с 3 параметъра
        // this.brand = brand;
        // this.model = model;
        // this.horsePower = -1;
    }

    public Car(String brand, String model, int horsePower) {
        this.brand = brand;
        this.model = model;
        this.horsePower = horsePower;
    }
}

```

Builder Pattern - Вариант за викане на конструктор когато се дублира сигнатурата -:

```

public class Engine {
    private String modelEngine;
    private int powerEngine;
    private String displacementEngine;
    private String efficiencyEngine;

    //конструктор
    public Engine(String modelEngine, int powerEngine) { //викане на Constructor Chaining
        this(modelEngine, powerEngine, "n/a", "n/a");
    }

    //правим метод - това е Builder Pattern
    public Engine EngineDisplacement(String modelEngine, int powerEngine, String
displacementEngine) {
        return new Engine(modelEngine, powerEngine, displacementEngine, "n/a");
    }

    //правим метод - Builder Pattern
    public Engine EngineEfficiency(String modelEngine, int powerEngine, String efficiencyEngine) {
        return new Engine(modelEngine, powerEngine, "n/a", efficiencyEngine);
    }
}

```

```

//конструктор
    public Engine(String modelEngine, int powerEngine, String displacementEngine, String
efficiencyEngine) {
        this.modelEngine = modelEngine;
        this.powerEngine = powerEngine;
        this.displacementEngine = displacementEngine;
        this.efficiencyEngine = efficiencyEngine;
    }
}

В метода main():
if (tokens.length == 3) {
    try { // когато имаме само displacement
        int displacementEngine = Integer.parseInt(tokens[2]); //ако не е int = displacement, то ще
e efficiency
        engine = new Engine(engineModel, Integer.parseInt(tokens[1]));
        engine = engine.EngineDisplacement(engineModel, Integer.parseInt(tokens[1]), tokens[2]);
    } catch (NumberFormatException e) { //когато имаме само efficiency
        engine = new Engine(engineModel, Integer.parseInt(tokens[1]));
        engine = engine.EngineEfficiency(engineModel, Integer.parseInt(tokens[1]), tokens[2]);
    }
}

```

3. Static Members

- Access static members **through the class name**
- Static members are **shared class-wide**
- You don't **need** an instance – no need to use **new** ... - статична променлива няма инстанция, с други думи не можем да използваме ключовата дума **this**

```
Main.main();
Math.abs();
```

Статичната променлива/поле е споделена между всички обекти/инстанции на класа!

```

public static class BankAccount {
    private static int idCounter = 0; //статично поле – споделен брояч
    private static double interestRate; //статично поле споделен лихвен процент за всички банкови
сметки

.....
public BankAccount() {
    this.id = BankAccount.idCounter;
    BankAccount.idCounter++;
    System.out.println("Account ID" + this.id + " created");
}
.....

```

Когато имаме статична локална клас-променлива в метода, то и метода го правим **static**

```

public static void setInterestRate(double interest) { //статичен метод
    BankAccount.interestRate = interest;
}
```

В main() метода:

BankAccount bankAccount = new BankAccount(); - увеличава BankAccount.idCounter с всеки нов обект на класа

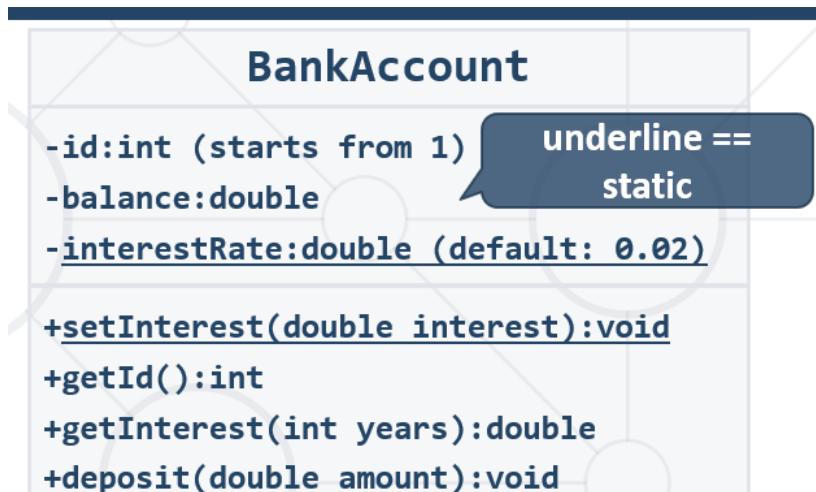
```
double newInterest = Double.parseDouble(tokens[1]);  
BankAccount.setInterestRate(newInterest); - задава нов лихвен процент за всички обекти/инстанции  
на класа/за всички депозити
```

4. UML diagram = Unified Modelling Language

Minus is private

Plus is public

Underlined is static



5. Other - всеки проект в нова папка в src

src/app/appFolders – в Java пишем проекти в папки – **всеки проект в нова папка в src**

Когато променяме стойност на обект, и ако този обект се намира в лист от обекти, то промяната се отразява и в листа от обекти!!!

При бази данни не работи така, и трябва да се презаписва наново!

Object(254)

List<Object>

(254)

object.setName("Gosho")

6. Types of class variables

In Java you can declare three types of variables namely, instance variables, static variables and, local variables.

- **Local variables** – Variables defined **inside methods, constructors or blocks** are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** – Instance variables are variables **within a class but outside any method**. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class (static) variables** – Class variables are variables declared **within a class, outside any method**, with the static keyword.

7. Shadowing in Java

```
// Outer Class
public class Shadowing {

    // Instance variable or member variable
    String name = "Outer John";

    // Nested inner class
    class innerShadowing {

        // Instance variable or member variable
        String name = "Inner John";

        // Function to print content of instance variable
        public void print()
        {
            System.out.println(name);
            System.out.println(Shadowing.this.name);
        }
    }

    // Driver Code
    public static void main(String[] args)
    {

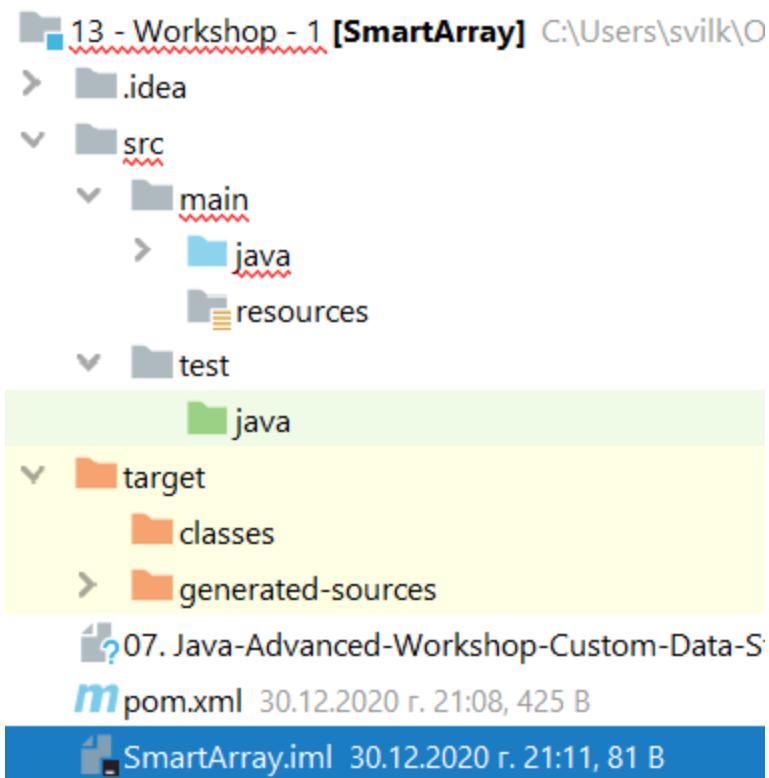
        // Accessing an inner class - как инициализираме обект от вътрешния нестнат клас
        Shadowing obj = new Shadowing();
        Shadowing.innerShadowing innerObj = obj.new innerShadowing();

        // Function Call
        innerObj.print();
    }
}
```

19. Разработване на клас – статична или динамична реализация - и Maven

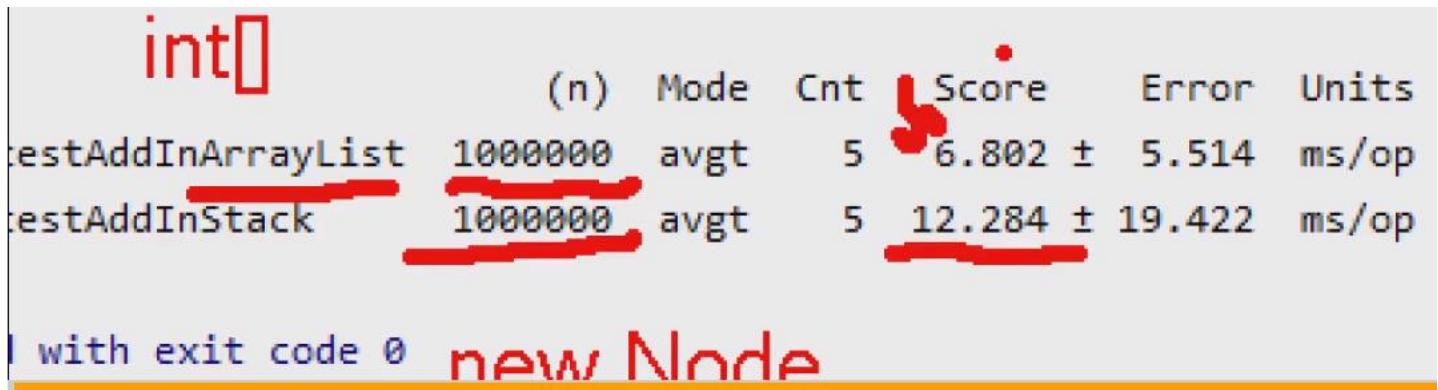
Maven е Project Management Tool

Target папката – за да не re-build-ва наново – пази последните build-и



Разлика между статичен подход и динамичен подход при разрешаване на проблем
Статична реализация – `int[] elements`

динамична реализация – Class Node или Stack с int element в него – думичката new заделя нови данни в паметта (с повече инструкции към процесора), и това е по-бавно от работа с масив (масива заделя по-рядко нова памет, по-рядко имаме new)



19.1. Статична реализация на SmartArray – задачата за симулиране на класа ArrayList с използване на масиви – кофти е

```
public class SmartArray {  
    private int[] elements;  
    private int index;  
  
    public SmartArray() {  
        this.elements = new int[8];  
        this.index = 0;  
    }  
  
    public void add(int element) {  
        //increase Length with 1 when we reach the current length  
        if (this.index == this.elements.length) {  
            this.elements = grow();  
        }  
  
        this.elements[index] = element;  
        index++;  
    }  
  
    private int[] grow() {  
        int[] newElements = new int[this.elements.length * 2];  
        System.arraycopy(this.elements, 0,  
                        newElements, 0, this.elements.length);  
        return newElements;  
    }  
  
    public int get(int index) {  
        ensureIndex(index);  
        return this.elements[index];  
    }  
  
    private void ensureIndex(int index) {  
        if (index >= this.size() || index < 0) {  
            throw new IndexOutOfBoundsException("SmartArray out of bounds for " +  
                "index " + index + " with size " + this.size());  
        }  
    }  
  
    public int size() {  
        return this.index;
```

```

}

public int remove(int index) {
    int element = get(index);

    for (int i = index; i <= this.size() - 2; i++) {
        this.elements[i] = this.elements[i + 1];
    }

    // this.size() - мястото, до което четеме записани елементи
    this.elements[this.size() - 1] = 0; //this.index винаги е с един повече
    this.index--; //намаляме мястото, до което четеме записи

    if (this.size() <= this.elements.length / 4) {
        this.elements = shrink();
    }

    return element;
}

private int[] shrink() {
    int[] newElements = new int[this.elements.length / 2];
    if (this.size() > 0) {
        System.arraycopy(this.elements, 0, newElements, 0, this.size());
    } else if (this.size() == 0) {
        this.elements = new int[8];
    }

    return newElements;
}

public boolean isEmpty() {
    return this.size() == 0;
}

public boolean contains(int element) {
    return this.indexOf(element) != -1;
}

public int indexOf(int element) {
    for (int i = 0; i < this.size(); i++) {
        if (element == this.elements[i]) {
            return i;
        }
    }

    return -1;
}

public void add(int index, int element) {
    int lastEl = this.get(this.size() - 1);

    for (int i = this.size() - 1; i > index; i--) {
        this.elements[i] = this.elements[i - 1];
    }

    this.elements[index] = element;
    this.add(lastEl);

    this.elements[index] = element;
}

```

```

public void forEach(Consumer<Integer> consumer) { //без използване на Iterable<> и Iterator<>
    for (int i = 0; i < this.size(); i++) {
        consumer.accept(this.elements[i]);
    }
}
}

```

19.2. Динамична реализация - ArrayDeque за стек или за опашка

Пример за динамична реализация на Stack – докато при ArrayDeque имаме индекси, до които нямаме достъп,

То вния пример нямаме индекси изобщо, а работим само с референции

```

public class MyStack {
    private Node top;
    private int size;

    public static class Node {
        private int element;
        private Node previous;

        Node(int element) {
            this.element = element;
            this.previous = null;
        }
    }

    public MyStack() {
    }

    public void push(int element) {
        Node newNode = new Node(element);
        if (this.top == null) {
            this.top = newNode;
        } else {
            newNode.previous = this.top;
            this.top = newNode;
        }
        this.size++;
    }

    public int peek(){
        this.ensureNotEmpty();

        return this.top.element;
    }

    public int pop(){
        this.ensureNotEmpty();
        int result = this.top.element;

        this.top = this.top.previous;
        this.size--;

        return result;
    }

    private void ensureNotEmpty() {

```

```

        if (this.top == null) {
            throw new IllegalStateException("Empty Stack");
        }
    }

    public int size(){
        return this.size;
    }

    public void forEach(Consumer<Integer> consumer){ //без използване на Iterable<> и Iterator<>
        Node current = this.top;

        while (current != null){
            consumer.accept(current.element);
            current = current.previous;
        }
    }
}

```

19.3. Динамична реализация - DoublyLinkedList

```

public class DoublyLinkedList {
    private ListNode head;
    private ListNode tail;
    private int size;

    private class ListNode {
        private int value;
        private ListNode next;
        private ListNode previous;

        public ListNode(int value) {
            this.value = value;
        }
    }
    ...
}

```

19.4. ArrayList и LinkedList

List<Integer> sample = new LinkedList<>(); - върши същата работа като ArrayList<>()

How the ArrayList works – **статична имплементация при класа ArrayList – масив отзаде**

The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

Класическият пример за LinkedList е за динамична имплементация при класа LinkedList – всеки елемент знае кой е предходния, но знае и кой е следващия! – референции отзаде

В Java е двойно свързана опашка със **статична имплементация – масив отзаде стои – за по-бърза работа/обработка на данните – не**

How the LinkedList works –

The LinkedList stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

19.5. Реализация на Hash функция и HashMap асоциативен масив

.hashCode()

Hash функцията винаги за един и същи вход връща един и същи изход – детерминистичен/deterministic approach подход. Или с други думи – за един и същи вход String връща един и същи Integer от паметта на даден компютър. Или за всяка една сесия на програмата ни, за едни и същи входни String, връща едни и същи Integer от паметта.

Също така, Integer стойността на даден Hash код с модулно делене, винаги дава един и същи резултат число.

Голямата цена на HashMap е, че всеки път се пре-разпределят елементите измежду броя buckets.

20. Generics - Adding Type Safety and Code Reusability

Java е статично типизиран език – предпазва ни от грешки.

Когато пишем Generic, първо си пишем с тип данни Integer или String например, и след това нагаждаме към Generic

20.1. General info – шаблон – в зависимост от типа данни, които му дадем, то трябва да може да работи

```
List strings = new ArrayList(); - нетипизиран – може да добавяме данни от различен тип  
strings.add("asd");  
strings.add(13);  
strings.add(true);  
strings.add(25.87);
```

List<Integer> numbers = new ArrayList<тук няма нужда да пишем пак Integer>(); - типизиран, може да добавяме данни само от тип Integer

Проблемът на Java 5 и надолу, е свързан с кастването и фактът, че не всичко може да се кастне експлицитно към даден тип (примитивни) данни.

20.2. Generic Classes

Type Parameter Scope - You can use it anywhere inside the declaring class

Defined with One type parameter:

```
public class Jar<Type> {  
    private Deque<Type> stack;  
  
    public Jar() {  
        this.stack = new ArrayDeque<>();  
    }  
  
    public void add(Type element){  
        this.stack.push(element);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Jar<Integer> jarInteger = new Jar<>(); - работи с Integer  
        Jar<String> jarString = new Jar<>(); - работи със String  
    }  
}
```

Defined with Multiple type parameters (2 type parameters):

```
class HashMap<K, V> {  
    /* magic */  
}
```

```

Класовете се екстендуват - Subclassing Generic Classes - Can extend to a concrete class -
class JarOfPickles extends Jar<Pickle> {
    ...
}

JarOfPickles jar = new JarOfPickles();
jar.add(new Pickle());
jar.add(new Vegetable()); // Error

```

20.3. Generic static and non-static Methods

Специфика на **static** синтаксиса – слагаме <E> преди типа данни и след това (типа на връщаните данни или void), които връща метода

ВАЖНО: Когато използваме **Generic масиви**, не винаги можем да използваме долния код. В такива ситуации, използваме масив от обекти (**Object[]**) вместо generic на места.

Как създаваме Generic масив в Java

```

package genericArrayCreator_02;

public class ArrayCreator { // може класа да няма Generic, само някои методи от класа да имат
    //статичните Generic методи реферират типа static <Type>,
    // а Type[] връща типа масива от кой тип е

Tree<E>[] longestPath = (Tree<E>[]) new Object[0];

    @SuppressWarnings("unchecked")
    public static <Type> Type[] create(int length, Type value) {
        //Type[] arr = (Type[]) new Object[length];
        Type[] arr = (Type[]) Array.newInstance(value.getClass(), length);

        for (int i = 0; i < length; i++) {
            arr[i] = value;
        }

        return arr;
    }

    public static <Type> Type[] create(int length, Type value) {
        return create(value.getClass(), length, value);
    }

    @SuppressWarnings("unchecked")
    public static <T> T[] create(Class<?> clazz, int length, T value){ //? WildCard - знае че е
обект, може да използваме и <T> вместо wildcard <?>
        T[] arr = (T[]) Array.newInstance(clazz, length);

        IntStream.range(0, length)
            .forEach(i -> arr[i] = value);

        return arr;
    }
}

```

```

public class Main {
    public static void main(String[] args) {
        String[] javas = ArrayCreator.<String>create(13, "Java"); // с 2 параметъра
        Integer[] integers = ArrayCreator.<Integer>create(Integer.class, 13, 69); //с 3
параметъра

        System.out.println();
    }
}

```

Да не създаваме Generic масиви за момента – очаква се в Java да оправят проблема. Да не създавам, за да не се наложи след време ръчно там където сме го използвали да го заменяме с новия бъдещ начин

Други - //статичните Generic методи реферират тук static <E>

```

public static <E> void swapElements(List<E> list, int firstIndex, int secondIndex){
    E firstElement = list.get(firstIndex);
    E secondElement = list.get(secondIndex);

    list.set(firstIndex, secondElement);
    list.set(secondIndex, firstElement);
}

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(reader.readLine());

        List<Box<String>> boxes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            Box<String> box = new Box<>(reader.readLine());
            boxes.add(box);
        }

        int[] indexes =
        Arrays.stream(reader.readLine().split("\s+")).mapToInt(Integer::parseInt).toArray();

        swapElements(boxes, indexes[0], indexes[1]);
    }
}

```

20.4. Generic Interfaces

Generic interfaces are similar to generic classes – класовете имплементират интерфейсите
Класовете могат да имат private и public методи, както и инстанции/обекти на класа
Интерфейсите нямат private методи, а само public, и не можем да имаме инстанция/обект от/на interface

```

interface List<T> {
    void add (T element);
    T get (int index);
    ...
}

class MyList implements List<MyClassType> {...} - клас MyList без Generic
class MyList<T> implements List<T> {...} - клас MyList<T> с Generic

List<MyClassType> - е интерфейс
public interface List<E> extends Collection<E> {} - е интерфейс

```

20.5. Type Erasure – типово изтриване

Generics are compile time illusion – Generics типовете са само и единствено за Safety по време на компилиране, по време на изпълнение реално няма Generics. Всичко става Object, реално като нямаме грешки при компилация благодарение на generics, то и по време на Runtime няма да има грешка.
Compiler deletes all angle bracket syntax – компилаторът изтрива всички тези скоби <>
Adds type casts for us (presented in byte-code)

След компилация, задрасканите все едно изчезват – разбираме го като използваме instanceof

```
List<String> strings = new ArrayList<String>();
```

1. System.out.println(strings instanceof List); //връща true – класа List<> наследява класа List,
и скобите <> ги няма

2. System.out.println(strings instanceof List<String>); //compile time error – illegal generic
type – и двете List и List<String> са едно и също

3.

```
strings.add("Pesho");
System.out.println(strings.get(0) instanceof String); //връща true
```

Type Erasure – Example – T не съществува след компилацията

```
public class Illusion<T> {
    public void function(Object obj) {
        if (obj instanceof T) {} // Error
        T[] array = new T[1]; // Error
        T newInstance = new T(); // Error
        Class cl = T.class; // Error
    }
}
```

20.6. Type Parameter Bounds

Generics позволява ограничението на типа данни да бъде само чрез extends на някой по-базов клас (за класове), и extends на някой по-базов интерфейс (за интерфейси)!

Не мога да задам като ограничение в generics скобите <> клас, който да имплементира интерфейс!

<T extends Animal>

Клас Cat екстендува класа Animal

<T extends Class> - specifies an "Upper/parent bound" – class Cat<T extends Animal>

В този случай, T е клас, наследник/дете на класа животно

Имам списък от животни, независимо какъв тип ще подам на животното, то ще е от базовия клас Animal

```
package upperBound;
```

```
public class Main {
    public static void main(String[] args) {
        AnimalList<Animal> animals = new AnimalList<>();
        AnimalList<Car> cars = new AnimalList<>(); //не може да създадем лист от Car, защото класа
Car не екстендува класа Animal.
        animals.add(new Cat("Maqu", 13)); - добавяме котка
        animals.add(new Animal("Animal", 42)) - добавяме обект от базовия клас Animal

        animals.printNames();
    }
}
```

```

public class Animal {
    private int age;
    private String name;

    public Animal(String name, int age){
        this.name = name;
        this.age = age;
    }

    public int getAge() { return age; }

    public String getName() { return name; }
}

public class Cat extends Animal {
    public Cat(String name, int age) {
        super(name, age);
    }
}

public class AnimalList<T extends Animal> extends ArrayList<T> {
    public void printNames() {
        Iterator<T> iterator = iterator();

        while (iterator.hasNext()){
            T next = iterator.next();
            System.out.println(next.getName()); - има достъп до всички методи на Animal класа
//                next.getAge();
        }
    }
}

```

<T extends Comparable<T>>

Екстендане на интерфейс с опция за сравняване. Тук интерфейса се екстенда – стандартното сравняване прави тук

```

public interface Comparable<T> {}

public class Scale<T extends Comparable<T>> - в този случай T е също интерфейс, от тип интерфейс Comparable<T>, но интерфейсът T се имплементира в класовете String и Integer реално
public class Main {
    public static class Scale<T extends Comparable<T>> {
        T left;
        T right;

        Scale(T left, T right) {
            this.left = left;
            this.right = right;
        }

        T getHeavier() {
            int result = this.left.compareTo(this.right); //0 -1 1 - същото като при класическо
//възходящо .sorted(), който използва съответно функцията Comparator<> first.compareTo(second);
!!! там ако е 1 има размяна = първи елемент по-голям от втори, ако е -1 няма размяна на
елементите = първи елемент по-малък от втори

            if (result == 0) {
                return null;
            }
        }
    }
}

```

```

    }
    if (result > 0) { // при по-голямо от нула, първият елемент е по-голям
        return this.left;
    }

    return this.right; // при по-малко от нула, вторият елемент е по-голям
}

public static void main(String[] args) {
    Scale<String> scale = new Scale<>("a", "z");
    System.out.println(scale.getHeavier());

    Scale<Integer> scale1 = new Scale<>(13, 7);
    System.out.println(scale1.getHeavier());
}
}

```

Когато класът съдържа единичен обект и compareTo сравнява един единичен обект с друг единичен обект
Имплементираме метода compareTo по наш си начин

```

public class Box<E extends Comparable<E>> implements Comparable<E>{
    private E data;

    @Override
    public int compareTo(E o) {
        return data.compareTo(o);
    }
}

```

20.7. Type Parameters Relationships

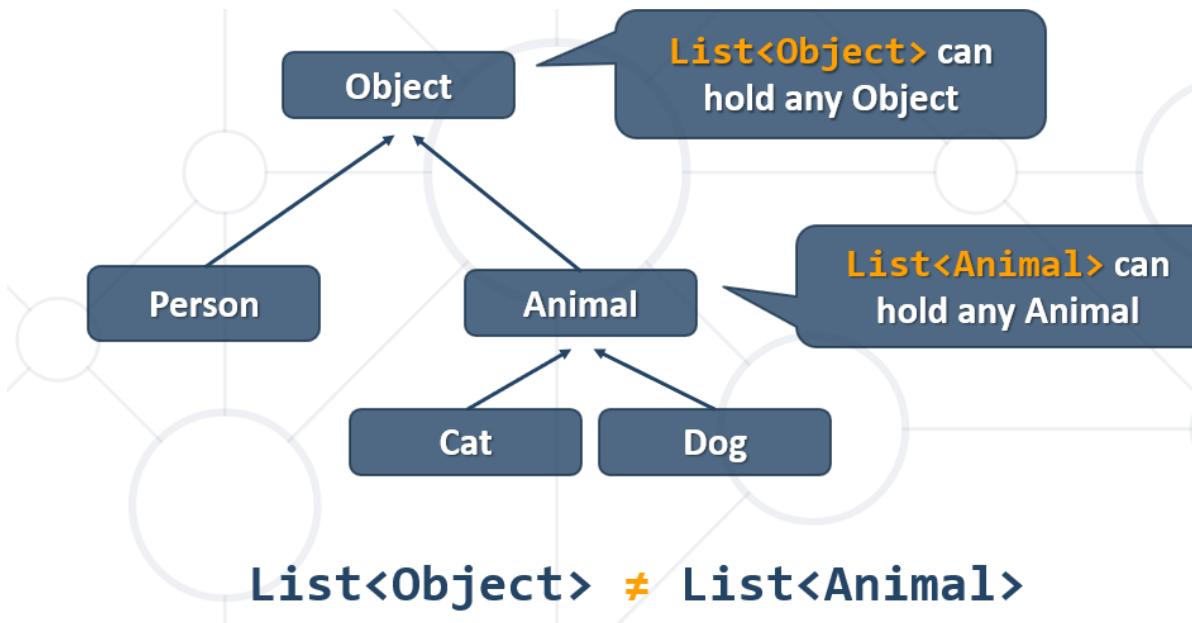
Generics are **invariant** = **never changing** - един обект/клас от Generics не може да присвои друг ако е от различен тип

```

List<Object> objects = new ArrayList<>();
List<Animal> animals = new ArrayList<>();
objects = animals; // Compile Time Error!

```

Списъкът от обекти може да е и Person!!!



20.8. Повече от един Generic

```
public class Tuple<Item1, Item2> {  
    private Item1 item1;  
    private Item2 item2;  
  
    public Tuple(Item1 item1, Item2 item2) {  
        this.item1 = item1;  
        this.item2 = item2;  
    }  
  
    public Item1 getItem1() {  
        return item1;  
    }  
  
    public Item2 getItem2() {  
        return item2;  
    }  
}
```

21. Iterators and Comparators

21.1. Variable Arguments (Varargs) – вариращ брой параметри в сигнатурата на метод

Сигнатура вариращ брой елементи – нула, един или повече на брой елементи

Unsafe операция е varArgs

```
public class Main {  
    public static void main(String[] args) {  
        printVarArgs();  
        printVarArgs("1", "2", "3");  
    }  
  
    public static void printVarArgs(String... elements) {  
        if (elements.length == 0) {  
            System.out.println("No elements");  
        } else {  
            String[] strings = elements; // масив  
            for (String string : strings) {  
                System.out.println(string);  
            }  
        }  
    }  
}  
  
private List<String> list;  
  
public ListyIterator(String... elements) {  
    this.list = List.of(elements); // лист  
}
```

Метод за дължина на varArgs
elements.length

В червено - въвеждане на varArgs

```
Book bookOne = new Book("Animal Farm", 2003, "A", "Terry Pratchet", "Tolkein", "Ivan Vazov");
Book bookOne = new Book("Animal Farm", 2003,
new String[]{"A", "Terry Pratchet", "Tolkein", "Ivan Vazov"});
```

Variable Arguments Rules: - 0 или повече от 0 входове на данни – VarArgs трябва да използват само един тип данни, ако е String... то може да представим стринга като числа или каквото е необходимо

- There can be only one variable argument in the method
- Variable argument must be the last argument

След VarArgs не може да има други параметри.

```
public static void printVarArgs(String... elements, int num) { }
```

Но може да има параметри преди VarArgs

```
public static void printVarArgs(boolean isTrue, int num, String... elements) { }
```

```
void method(String... a, int... b){} //Compile time error – не може да има 2 VarArgs параметри
void method(int... a, String b{}) //Compile time error – VarArgs не е последен
```

VarArgs с Generics

```
public static <T> void printVarArgs(T... elements) {
    if (elements.length == 0) {
        System.out.println("No elements");
    } else {
        T[] arrList = elements;
        for (T element : arrList) {
            System.out.println(element);
        }
    }
}
```

Това не е вариращи параметри:

Мога да стартирам програмата с никакви параметри/входни данни от отвън. Но трябва да има поне едно въвеждане на данни!!!! – 1 или повече входове на данни

```
public class Main {
    public static void main(String[] args) {
        printVarArgs();
        printVarArgs("1", "2", "3");
    }
}
```

При varArgs, конструкторът на даден клас може да се извика както с дефолтния нулев конструктор, така с конструктор с елементи

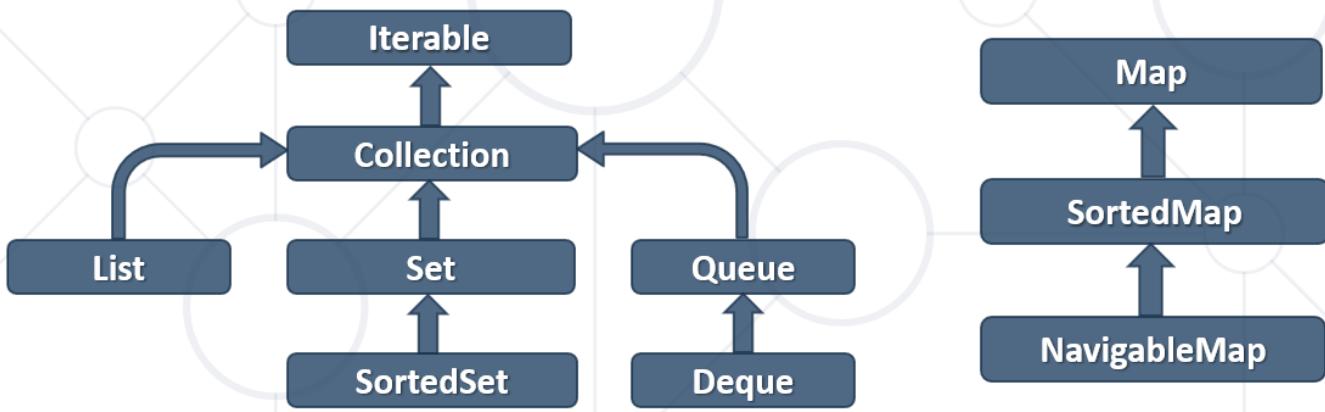
```
public class MessageLogger implements Logger {
    private Set<Appender> appenders;

    public MessageLogger(Appender... appenders) {
        this.setAppendlers(appenders);
    }

    public class Main {
        public static void main(String[] args) {
            Logger logger = new MessageLogger(); //нулев конструктор позволява!!!
        }
    }
}
```

21.2. Iterable<T> and Iterator<T> interfaces

Inheritance leads to **hierarchies** of classes and/or interfaces in an application:



1) *Iterable<T>* - казва ти - можеш да обходиш тази структура

Можем да **обходим елементите/полетата** на класа който имплементира *Iterable<E>*

```
public class Book implements Iterable<Book>{  
    private String title;  
    private int year;  
    private List<String> authors;  
  
    public Book(String title, int year, String... authors) {  
        this.setTitle(title);  
        this.setYear(year);  
        this.setAuhtors(authors);  
    }  
}
```

Iterable<T>

Root interface of the Java collection classes

A class that implements the *Iterable<T>* can be used with the new **for loop**

```
Book bookOne = new Book("Animal Farm", 2003, "A", "Terry Pratchet", "Tolkein", "Ivan Vazov");  
for (ObjectBook book : bookOne) {  
    //ще обходи името на книгата, годината на издаване и авторите ако има такива- реално е тъло да го  
    //използваме така public class Book implements Iterable<Book>{}  
}
```

```
List<Book> books = new ArrayList<>();  
books.add(bookOne);  
books.add(bookTwo);  
for (Book book : books) {  
    // ще обходи листа от книги, тъй като List наследява interface Collection, а interface Collection  
    //наследява interface Iterable<>  
}
```

Abstract methods of Iterable<T>

- **iterator()** – спомага за обхождане на елементите след това - с **iter (for)**.

```

public interface Iterable<T> {
    public Iterator<T> iterator(); //мимо деклариране
}

```

- Default methods

`forEach(Consumer<? super T> action)` //super ни ограничава да не объркаме типовете – нива нагоре може да сравняваме само – един вид WildCard; все едно *Integer super Number*; *super* – към бащин клас

- `spliterator()` - used for parallel programming

Пример:

Вместо `Iterable<E>` винаги можем да използваме `List<E>` за по-лесно.

Понякога се налага да използваме `List<E>` за да създадем колекция, която да върнем. И отвън можем да използваме обратно като `Iterable<E>`

```

@Override
public Iterable<E> find(Class<E> table, String where) throws SQLException, NoSuchMethodException,
IllegalAccessException, InvocationTargetException, InstantiationException {
    String tableName = getTableName(table);

    String selectQuery = String.format("SELECT * FROM %s %s", tableName,
        where != null ? "WHERE " + where : "");

    PreparedStatement statement = connection.prepareStatement(selectQuery);
    ResultSet resultSet = statement.executeQuery();

    List<E> output = new ArrayList<>();
    while (resultSet.next()) {
        E resultEntity = table.getDeclaredConstructor().newInstance();
        fillEntity(table, resultSet, resultEntity);
        output.add(resultEntity);
    }

    return output;
}

public class Main {
    public static void main(String[] args) throws SQLException, IllegalAccessException,
    NoSuchMethodException, InstantiationException, InvocationTargetException {
        MyConnector.createConnection("root", "", "custom-orm");
        Connection connection = MyConnector.getConnection();

        EntityManager<User> userEntityManager = new EntityManager<>(connection);

        User user = new User("Svilen", 36, LocalDate.now());

        Iterable<User> first = userEntityManager.find(User.class, "id<5");
        System.out.println(first);
    }
}

```

Разпечатката на резултата на `Iterable<User>` в конзолата е масив от клас-обекти:

```

first = {ArrayList@2600} size = 2
  0 = {User@2631} "User{id=2, username='pesho_new_new"
    id = 2
    username = "pesho_new_new"
    age = 25
    registrationDate = {LocalDate@2636} "2022-02-22"
    lastLoggedIn = {LocalDate@2637} "2022-02-22"
  1 = {User@2632} "User{id=3, username='Svilen', age=36,
    id = 3
    username = "Svilen"
    age = 36
    registrationDate = {LocalDate@2641} "2022-02-22"
    lastLoggedIn = {LocalDate@2642} "2022-02-22"

```

2) `Iterator<T>` - знае и обхожда съответната структура

Знае как да **обходим елементите/полетата** на класа който имплементира `Iterable<E>`

Enables you to cycle through a collection

Nested class for `Iterator<T>`

```

public class Library<T> implements Iterable<T> {
    private final class LibIterator implements Iterator<T> {}
}
```

```

public static void main(String[] args) {
    String[] authors = new String[]{"A", "Terry Pratchet", "Tolkein", "Ivan Vazov"};
    Book bookOne = new Book("Animal Farm", 2003, "A", "Terry Pratchet", "Tolkein", "Ivan Vazov");
    Book bookThree = new Book("The Documents in the Case", 2002);
    Book bookTwo = new Book("The Documents in the Case", 1930, "Dorothy Sayers", "Robert Eustace");

    List<Book> books = new ArrayList<>();
    books.add(bookOne);
    books.add(bookTwo);
    books.add(bookThree);

    Iterator<Book> iterator = books.iterator();
}
```

Don't implement both `Iterable<T>` and `Iterator<T>`

```
class MyClass implements Iterable<T>, Iterator<T> {} // да не правим така
```

Като си направим мним или реален итератор<>, можем да го обхождаме с `iter (for)`.

Вариант 1

при използване на `Iterator<T>` и отделен вътрешен клас, който обхожда

```

public class Library implements Iterable<Book> {
    private Book[] books;
    public Library(Book... books) {
        this.books = books;
    }
}
```

```

@Override // метод предизвикан от интерфейса Iterable<T>
public Iterator<Book> iterator() {
    return new LibIterator();
}

private class LibIterator implements Iterator<Book> {
    private int i = 0;

    @Override
    public boolean hasNext() {
        return i < books.length;
    }

    @Override
    public Book next() {
        return books[i++];
    }
}
}

```

Вариант 2

При използване на `Iterator<T>` - когато вътрешния клас е мним, и не може да го достъпваме, и е еднократен

```

@Override // метод предизвикан от интерфейса Iterable<T>
public Iterator<E> iterator() {
    return new Iterator<E>() { //това е мнимото създаване на вътрешния клас LibIterator
        int index = 0;
        @Override
        public boolean hasNext() {
            return index < list.size();
        }

        @Override
        public E next() {
            return list.get(index++);
        }
    };
}

```

Вариант 3

Когато използваме итератора на вече създадената структура

```

public class Library implements Iterable<Book>{
    private List<Book> books;

    @Override // метод предизвикан от интерфейса Iterable<T>
    public Iterator<E> iterator() {
        return this.books.iterator();
    }
}

```

Разглеждане на обикновен `Iterator<T>` на лист

```

public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    numbers.add(13);    numbers.add(42);
    numbers.add(69);    numbers.add(73);

    Iterator<Integer> iterator = numbers.iterator();
}

```

```

while (iterator.hasNext()){
    System.out.println();
}
}

```

Разглеждане на listIterator<> (лист-итератор) – има много повече функции – ползва се от List или LinkedList структурите

```

public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<>();
    numbers.add(13);    numbers.add(42);
    numbers.add(69);    numbers.add(73);

    ListIterator<Integer> iterator = numbers.listIterator();

    while (iterator.hasNext()){
        if(iterator.hasPrevious()) {}
        System.out.println(iterator.previous());
        System.out.println(iterator.previousIndex());
        iterator.add(5);
    }
}

```

21.3. Comparable<T> and Comparator<T>

1. Comparable<E>

Казва, че елементите на класа имплементиращ Comparable<> са сравними по някакъв естествен ред чрез метода compareTo.

Реално в метода compareTo ние си задаваме конкретна логика на сравнение, но трябва да можем без да гледаме тази логика в compareTo метода, да знаем какво върши. Ако се наложи да гледаме в compareTo метода за логиката, значи ни трябва Comparator<T> и метода compare! Така е прието в Java!

Означава, че всеки един обект на класа имплементиращ Iterable<T> ще се сравнява по този начин описан в метода compareTo. Докато с Iterator<T> и метода compare може да имаме много на брой различни класове имплементиращи Comparator<T> сравняващи някаква външна структура от данни по различни начини – във всеки от класовете метода compare да има различна логика.

Инстанции на нашия клас, който имплементира Comparable<E>, могат да се сравняват

Comparable allows you to specify how objects that you are implementing get compared

Single sorting sequence

Affects the original class

compareTo() method first.compareTo(second); сравнение този с другия по естествен ред

```

public class Book implements Comparable<Book>{
    private String title;
    private int year;

    @Override //– важи този начин на сортиране за всички chainings на бъдещия stream
    public int compareTo(Book other) {
        return this.title.compareTo(other.title);
        return Integer.compare(this.year, other.year);
    }
}

```

или

```

if (this.age == other.age) { return 0; }
else if (this.age > other.age) { return 1; }
else if (this.age < other.age) { return -1; }
}

```

}

first.compareTo(second);

Връща 1, -1 или 0

Ако получим резултат 1 – двете числа променят мястото си = има размяна = първото е по-голямо от второто

Ако получим резултат -1 – двете числа не променят мястото си = няма размяна = първото е по-малко от второто

Ако получим 0 - двете числа отново не променят мястото си/реда си

```
public static void main(String[] args) {  
    Book[] books = new Book[]{bookOne, bookTwo, bookThree};  
    Arrays.stream(books).sorted().forEach(System.out::println);  
}
```

2. Comparator<E>

Comparator provides a way for you to provide custom comparison logic for types that you have no control over
.sorted() използва функция Comparator<>

Multiple sorting sequence

Doesn't affect the original class

compare() method - сравнение левия с десния – подобен метод на compareTo – създаваме си собствена логика на сравнение за дадена структура от данни

Вариант 1: - ламбда мнимата функция и мним клас на Comparator<>

```
Book[] books = new Book[]{bookOne, bookTwo, bookThree};  
Arrays.stream(books).sorted((f, s) -> Integer.compare(f.getYear(), s.getYear())).forEach(System.out::println);  
Или с Comparator.comparingInt(entry -> entry.getYear) може също
```

.....като нищо не пишем в метода compareTo, тъй като проверката/логиката за сравнение се задава отвън, то не имплементираме нищо!!! – за да не се кара, задрасканото го пишем все пак!!!

```
public class Book implements Comparable<Book>{  
    @Override  
    public int compareTo(Book other) {  
        return 0;  
    }  
}
```

Вариант 2: показване на мнимата функция и мним клас на Comparator<> като видими – изнасяме класа имплементиращ Comparator<> в отделен файл/клас

```
public class CompareBooksByYearsAscending implements Comparator<Book> {  
    @Override  
    public int compare(Book first, Book second) {  
        return Integer.compare(first.getYear(), second.getYear());  
    }  
}  
или  
public class CompareBooksByYearsDescending implements Comparator<Book> {  
    @Override  
    public int compare(Book first, Book second) {  
        return Integer.compare(second.getYear(), first.getYear());  
    }  
}  
или
```

```

public class BookComparator implements Comparator<Book> {
    @Override
    public int compare(Book first, Book second) {
        int result = first.getTitle().compareTo(second.getTitle());
        return result != 0 ? result : Integer.compare(first.getYear(), second.getYear());
    }
}

```

Имаме от горното 3 класа за различен вид сортиране

```

public static void main(String[] args) {
    CompareBooksByYearsAscending compareBooksByYearsAscending = new CompareBooksByYearsAscending();
    Arrays.stream(books).sorted(compareBooksByYearsAscending::compare).forEach(System.out::println);
    Или
    CompareBooksByYearsAscending compareBooksByYearsAscending = new CompareBooksByYearsAscending();
    Arrays.stream(books).sorted(compareBooksByYearsAscending).forEach(System.out::println);
    Или само
    Arrays.stream(books).sorted(new CompareBooksByYearsAscending()).forEach(System.out::println);

    books.sort(new BookComparator());
}

```

Когато са валидни сортировките само за текущия chaining на stream-a.

Използваме или ламбда или съкратения вариант с Comparator.comparing

Ламбда функцията може да се запише като Comparator.comparing

```

Arrays.stream(books)
    .sorted((f, s) -> f.getTitle().compareTo(s.getTitle())).forEach(System.out::println);

```

Когато използваме с Comparator.comparing, то ни позволява да правим при равно, то следващо сравнение по друг критерий, и след това може да ревърснем

```

Arrays.stream(books).sorted(Comparator.comparing(Book::getTitle).thenComparing(Book::getYear)
    .reverse())
    .forEach(System.out::println);

```

Когато има само дефautна сортировка, и тя е валидна за който и да е chaining на stream-a.

Коригираме метода на Iterable<Book>, тогава сортировката ще работи винаги по този начин:

```

public class Book implements Comparable<Book>{
    @Override
    public int compareTo(Book other) {
        int result = this.title.compareTo(other.title);
        return result != 0 ? result : Integer.compare(this.year, other.year);
    }
}

.stream()
.sorted((product, other) -> product.compareTo(other));

.stream()
.sorted(Product::compareTo);

```

```

List<Book> books = new ArrayList<>(Arrays.asList(bookOne, bookTwo, bookThree));
Collections.sort(books);

```

Има и вариант с добавяне на логика за сравнение в Collections.sort, което допълнение прави същото нещо:
Collections.sort(books, (f, s) -> f.compareTo(s));

Компараторите **Comparator<>** са изключително полезни когато работим с TreeSet или TreeMap, тъй като може да се използва за моментално подреждане докато се създава структурата данни, и по-малко операции/ресурси за процесора в сравнение с последваща сортировка!!!

Set<Book> bookSet = new TreeSet<>(new BookComparator()); - иначе няма да знае TreeSet-а как да ги сравни!!!

```
Set<Person> byAge = new TreeSet<>(new CompareByAge());
public class CompareByAge implements Comparator<Person> {
```

```
    @Override
    public int compare(Person o1, Person o2) {
        return Integer.compare(o1.getAge(), o2.getAge());
    }
}
```

Същото като

```
Set<Book> bookSet = new TreeSet<>();
bookSet.sort(new BookComparator());
..
Set<Person> byAge = new TreeSet<>(); // това не работи
byAge.sort(new CompareByAge()); // това не работи
```

Comparing object with Equals() and hashCode():

1. Whenever you implement equals, you MUST also implement hashCode
2. For example, the Strings "Aa" and "BB" produce the same hashCode: 2112. Therefore: Never misuse hashCode as a key
3. Do not use hashCode in distributed applications
4. Best advice is probably: don't use hashCode at all, except when you create hash-based algorithms.

equals() compares the objects' fields. **If two objects have the same field values, then the objects are the same.**

Лесен начин за извикване на метода equals и метода hashCode – Alt + Insert

Пример 1:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true; //извиква обекта на класа - реално чрез метода hashCode / и
    // сравняваме по референция
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return age == person.age && Objects.equals(name, person.name);
}

@Override
public int hashCode() { //генерира се hashCode за всеки обект - когато hashCode съвпада, то обекта
    // е същият в голяма част от случаите, но не всички!!!!
    return Objects.hash(name, age);
}
```

Пример 2 - за сравняване по hashCode():

```
@Override
public int hashCode() {
    return Integer.hashCode(this.id) * 17; //инстанцията на класа приема hashCode
}

@Override
public boolean equals(Object obj) {
    //в по-стари версии правим проверка и за null
    if (!(obj instanceof TransactionImpl)) { //обектът е инстанция на класа TransactionImpl ли е?
        return false; //ако не е, то върни false
    }

    TransactionImpl other = (TransactionImpl) obj;
    return this.hashCode() == other.hashCode(); //обектите на класа се сравняват по hashCode
    return this.id == other.id; //обекта се сравнява по полето ID
}
```

Пример 3:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true; //извиква обекта на класа - реално чрез метода hashCode / и
    //сравнява по референция???? Сравнява по референция в паметта

    //Подробната проверка с равно и equals дали и полетата на двата обекта са равни
    if (!(o instanceof WizzardDeposits)) return false;

    WizzardDeposits that = (WizzardDeposits) o;
    return getId() == that.getId() &&
           getFirstName().equals(that.getFirstName()) &&
           getLastname().equals(that.getLastname());
}

@Override //Ако hash кода е различен, то със сигурност няма нужда да правим проверка equals тъй
//като ще са различни. Но ако hashCode е еднакво число, то е възможно обектите или да са равни или
//различни. Тогава има нужда да проверим и по елементите на обекта дали си съвпадат!
public int hashCode() { //бързата проверка по hashCode
    return Objects.hash(getId(), getFirstName(), getLastname());
}
```

22. Functional programming - expressions

22.1. Lambda Expressions

Expression е всичко което връща резултат – или връща тип данни или връща void

Lambda Expression **is unnamed function**

Една нормална функция(винаги метод за Java) в Java съдържа следните елементи:

- 1) Return type**
- 2) Function name**
- 3) Parameters**
- 4) Body**

Една ламбда функция в Java съдържа (може да съдържа) само:

- 3) Parameters**
- 4) Body**

Lambda Syntax

(parameters) -> {body}

-> значи goes to

Implicit lambda expression:

(msg) -> { System.out.println(msg); } - Parameters can be enclosed in parentheses (), The body can be enclosed in braces {}

Explicit lambda expression:

String msg -> System.out.println(msg); - Declares parameters' type

Can have different number of parameters:

- Zero parameters

```
() -> { System.out.println("Hello!"); }  
() -> { System.out.println("How are you?"); }
```

- More parameters

```
(int x, int y) -> { return x + y; }  
(int x, int y, int z) -> { return (y - x) * z; }
```

Пример за анонимна и видима функция - 1

```
List<Integer> numbers = List.of(50, 12, 48);  
- Анонимна ламбда функция  
numbers.forEach(e -> System.out.println(e));  
numbers.forEach(Integer num -> System.out.println(num));
```

- Функция видима

```
numbers.forEach(new Consumer<Integer>() {  
    @Override  
    public void accept(Integer num) {  
        System.out.println(num);  
    }  
});
```

Пример за анонимна и видима функция - 2

- Анонимна ламбда функция

```
int[] numbers = Arrays.stream(sc.nextLine().split(", ")).  
    .mapToInt(num -> Integer.parseInt(num))
```

- Функция видима

```
int[] numbers = Arrays.stream(sc.nextLine().split(", ")).  
    .mapToInt(newToIntFunction<String>() {  
        @Override  
        public int applyAsInt(String value) {  
            return Integer.parseInt(value);  
        }  
    })  
.toArray();
```

Други

Използване на IntConsumer

```
int[] numbers = Arrays.stream(sc.nextLine().split(", ")).  
.mapToInt(num -> Integer.parseInt(num))
```

```

.filter(num -> num % 2 == 0)
.toArray();
//приема mun, но не връща параметри
IntConsumer consumer = num -> System.out.println(num + " ");
Arrays.stream(numbers).forEach(consumer);

Използване на Consumer<int[]>
int[] numbers = Arrays.stream(sc.nextLine().split(", "))
    .mapToInt(num -> Integer.parseInt(num))
    .filter(num -> num % 2 == 0)
    .toArray();

Consumer<int[]> consumer =
(arr) -> System.out.println(Arrays.stream(numbers)
    .mapToObj(num -> Integer.toString(num)) // прави го от число на стринг
    .collect(Collectors.joining(", ")));

consumer.accept(numbers);

```

Печатаме числа разделени с ", " чрез използване на функционално програмиране / в режим на stream

```

System.out.println(Arrays.stream(numbers)
    .mapToObj(num -> String.valueOf(num)) // прави го от число на стринг
    .collect(Collectors.joining(", ")));

```

Метод референция (method reference)

```
Consumer<String> printer = System.out::println;
```

С Ламбда израз: Consumer<String> printer = str -> System.out.println(str);

Мутация на елементите:

```

Consumer<String> printer = e -> System.out.println();
Arrays.stream(sc.nextLine().split("\s+"))
    .map(e -> "Sir " + e)
    .forEach(e-> printer.accept(e));

```

22.2. Functions (Mathematical and Java)

Едно от важните предимства на функциите в Java, е че можем да ги подаваме като параметър на други функции. Също така нито една функция няма състояние / stateless е.

Ако искаме да подадем един метод на друг метод, то винаги се обръщаме към обектно-ориентираното програмиране и към класа, от който е всеки метод. Обектите/инстанциите винаги имат състояние.

In Java **Function<T,R>** is an interface that accepts a parameter of type **T** and returns variable of type **R**

```
Function<Integer, Integer> func = x -> x * x;
```

Функция, която използваме в stream Api – Пример 1

```

ToIntFunction<String> parsInt = x -> Integer.parseInt(x);
int[] numbers = Arrays.stream(sc.nextLine().split(", "))
    .mapToInt(parsInt)
    .toArray();

```

Функция, която използваме в stream Api – Пример 2

```

Function<String, Integer> parse = e -> Integer.parseInt(e);
Integer[] numbers = Arrays.stream(sc.nextLine().split(", "))
    .map(parse)

```

```
.filter(num -> num % 2 == 0)
.toArray(Integer[]::new); - бащин клас масив
```

We use function with `.apply()`

`Function<Integer, Integer> squared = x -> x * x;` - първият е входните данни, втория параметър е типа на изхода

```
System.out.println(squared.apply(25)); //връща 625
```

Пример без chain-ване:

```
String[] tokens = sc.nextLine().split(", ");
Function<String[], Stream<Integer>> mapFunct = arr -> Arrays.stream(arr).map(Integer::parseInt);

Function<Stream<Integer>, Long> count = str -> str.mapToInt(e -> e).count(); - от бащин към примит
Function<Stream<Integer>, Integer> sum = str -> str.mapToInt(e -> e).sum();

Stream<Integer> stream = mapFunct.apply(tokens);
System.out.println("Count = " + count.apply(stream));

stream = mapFunct.apply(tokens);
System.out.println("Sum = " + sum.apply(stream));
```

Композиция/chain от функции

```
Function<Integer, Integer> squared = x -> x * x;
Function<Integer, Integer> multiplyByThree = x -> x * 3;

Integer result1 = squared.compose(squared).apply(4); //andThen()
Integer result2 = squared.compose(multiplyByThree).apply(4); // 12 * 12 = 144
System.out.println(result1);
System.out.println(result2);
```

С chain-ване

```
Function<int[], int[]> printCount = arr -> {
    System.out.println("Count = " + arr.length);
    return arr;
};
Function<int[], String> formatArrSum = arr -> "Sum = " + Arrays.stream(arr).sum();
System.out.println(printCount.andThen(formatArrSum).apply(numbers)); //първо се изпълнява apply, и
след това andThen
```

Без chain

```
Function<int[], Integer> printCount = arr -> arr.length;
Function<int[], String> formatArrSum = arr -> "Sum = " + Arrays.stream(arr).sum();
System.out.println("Count = " + printCount.apply(numbers));
System.out.println(formatArrSum.apply(numbers));
```

22.3. Function Types except the main type of one input and one output

`Consumer<T>` - void interface / приема нещо и прави нещо

In Java `Consumer<T>` is a void interface:

T-то е входен параметър тук

```
Consumer<String> printer = str -> System.out.println(str);
```

```
Consumer<String> printer = System.out::println;
Arrays.stream(sc.nextLine().split("\\\\s+")).forEach(printer);
Arrays.stream(sc.nextLine().split("\\\\s+")).forEach(e -> printer.accept(e));
```

```
void print(String message) {
    System.out.println(message);
}
```

We use a Consumer with .accept():

```
Consumer<String> print = message -> System.out.print(message);
print.accept("Peter");
```

Когато един метод, примерно за обхождане на данни в цикъл, всеки път може да прави различни неща, то няма нужда да има много методи с подобни имена forEach1, forEach2, а използваме **Consumer<Integer>**, и Ламбда функцията казва какво се прави с всеки елемент – тук е от бащин тип входния параметър

```
public void forEach(Consumer<Integer> consumer){
    for (int i = 0; i < this.size(); i++) {
        consumer.accept(this.elements[i]);
    }
}
```

Когато имаме метод с име, то може да викаме метода с ИМЕ, който да се изпълнява върху всеки елемент от **Consumer<Integer>**

```
public static void main(String[] args) {
    SmartArray smartArray = new SmartArray();
    for (int i = 1; i <= 8; i++) {
        smartArray.add(i);
    }
    smartArray.forEach(Main::print);
}

public static void print(int element){
    System.out.println(element);
}
```

Supplier<T> - не приема нищо без входни параметри, но връща/взема изход

In Java **Supplier<T>** takes no parameters: - не приема вход, но взема изход

T-то е типът, който ще бъде върнат

```
int genRandomInt() {
    Random rnd = new Random();
    return rnd.nextInt(51);
}
```

We use a Supplier with .get():

```
Supplier<Integer> genRandomInt = () -> new Random().nextInt(51);
int rnd = genRandomInt.get();
```

Predicate<T> - приема нещо, и връща true или false

In Java **Predicate<T>** evaluates a condition: - метод, който връща true или false

T-то е също входен параметър тук

```
boolean isEven(int number) {
    return number % 2 == 0;
}
```

We use the Predicate with `.test()`:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(13);
numbers.add(6);
numbers.add(7);
numbers.add(8);

Predicate<Integer> isEven = x -> x % 2 == 0;

numbers.stream()
    .filter(isEven); или .filter(s -> isEven.test(s)) или .filter(isEven::test)
```

How to easily produce predicates:

```
Predicate<String> predicate = producePredicates(stringCommand, stringParam);
private static Predicate<String> producePredicates(String command, String param) {
    Predicate<String> check = null;
    switch (command) {
        case "StartsWith": check = str -> str.startsWith(param);
        break;
        case "EndsWith": check = str -> str.endsWith(param);
        break;
        case "Length": check = str -> str.length() == Integer.parseInt(param);
        break;
    }
    return check;
}
```

Хубав пример за предикат – дали цикълът да върти от малко към голямо или обратно.

```
private static String printMultipleRows(int start, int end, int step, int size) {
    StringBuilder outx = new StringBuilder();
    Predicate<Integer> loopCondition = itt -> {
        if (step > 0) {
            return itt <= end;
        }
        return itt >= end;
    };

    for (int line = start; loopCondition.test(line); line += step) {
        outx.append(printLine(size - line, line)).append(System.lineSeparator());
    }

    return outx.toString();
}

private static String buildRombOfStars(int size) {
    StringBuilder sb = new StringBuilder();
    sb.append(printMultipleRows(1, size, +1, size))
        .append(printMultipleRows(size - 1, 1, -1, size));
    // for (int Line = 1; Line <= size; Line++) {
    //     sb.append(printLine(size - Line, Line)).append(System.lineSeparator());
    // }
    //
    // for (int Line = size - 1; Line >= 1; Line--) {
    //     sb.append(printLine(size - Line, Line)).append(System.lineSeparator());
    // }

    return sb.toString();
}
```

Комбинация от Supplier<T> и Predicate<T>

```
public class Test {
    public static Supplier<List<Person>> generateRandPeople = () -> {
        List<Person> people = new ArrayList<>();
        for (int i = 0; i < 100; i++) {
            people.add(new Person(Character.toString(i), new Random().nextInt(100)));
        }
        return people;
    };

    public static class Person {
        private String id;
        private int age;

        public Person(String ID, int age) {
            this.id = id;
            this.age = age;
        }
    }

    public static List<Person> getByPredicate(Collection<Person> coll, Predicate<Person> predicate) {
        return coll.stream().filter(predicate).collect(Collectors.toList());
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<Person> people = generateRandPeople.get();
        List<Person> collect = people.stream().filter(person -> person.age < 19).collect(Collectors.toList());
        collect = getByPredicate(people, p -> p.age >= 19);
    }
}
```

UnaryOperator - Функция, която приема и връща един и същи тип стойност и се изпълнява само върху един параметър

```
UnaryOperator<Double> addVAT = price -> price * 1.20;
Arrays.stream(sc.nextLine().split(", ")).mapToDouble(str -> addVAT.apply(Double.parseDouble(str))).forEach(vat -> System.out.println(String.format("%.2f", vat))));
```

Comparator<T>

T-то е типа на входен параметър тук

```
Predicate<Person> predicate = z -> z.getAge() > 30;
Comparator<Person> comparat = (f, s) -> s.getName().compareTo(f.getName());
Consumer<Person> consumer = x -> System.out.print(String.format("%s - %d", x.getName(), x.getAge()));

people.stream()
    .filter(predicate)
    .sorted(comparat)
    .forEach(consumer);
```

Iterator<T>

T-то е типа на входен параметър тук

```

List<Integer> numbers = new ArrayList<>();
numbers.add(13);    numbers.add(42);
numbers.add(69);    numbers.add(73);

Iterator<Integer> iterrr = numbers.iterator();

while (iterrr.hasNext()){
    System.out.println();
}

```

22.4. Bi Functions

Using Functions With More Parameters

BiFunction <T, U, R>

Input parameters are T and U

Output parameter is R

```

BiFunction<Integer, Integer, String> sum = (x, y) -> "Sum is" + (x + y);

Function<int[], Integer> minFunction = arr -> {
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < arr.length; i++) {
        if (min > arr[i]) {
            min = arr[i];
        }
    }
    return min;
};

minFunction.apply(Arrays.stream(sc.nextLine().split("\\s+"))
    .mapToInt(Integer::parseInt)
    .toArray());

```

Пример за функция, като различните входни параметри ги слагаме в Map

```

List<Integer> numbers = Arrays.stream(sc.nextLine().split("\\s+"))
    .map(Integer::parseInt)
    .collect(Collectors.toList());
Map<String, Function<List<Integer>, List<Integer>>> functionMap = new HashMap<>();
functionMap.put("add", e -> e.stream().map(val -> val +1).collect(Collectors.toList()));
functionMap.put("multiply", e -> e.stream().map(val -> val * 2).collect(Collectors.toList()));
functionMap.put("subtract", e -> e.stream().map(val -> val - 1).collect(Collectors.toList()));
functionMap.put("print", e -> e.stream().peek(z-> System.out.print(z + " "))
    .collect(Collectors.toList()));
numbers = functionMap.get(input).apply(numbers);

```

Analogically you can use:

BiConsumer <T, U> - два входни параметъра от тип T и U, и връща void

BiPredicate <T, U> - два входни параметъра от тип T и U, и връща boolean

```

public class ListOfPredicates {
    public static BiPredicate<Integer, Integer> predicate = (f, s) -> f % s == 0;

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

```

```

int n = Integer.parseInt(sc.nextLine());
Set<Integer> numbers = Arrays.stream(sc.nextLine().split("\\s+"))
    .map(Integer::parseInt)
    .collect(Collectors.toSet());

checkNumbers(1, numbers, n);
}

private static void checkNumbers(int num, Set<Integer> numbers, int n) {
    if (num > n) {
        return;
    }

    boolean isValid = true;
    for (Integer number : numbers) {
        if (!predicate.test(num, number)) {
            isValid = false;
            break;
        }
    }

    if (isValid) {
        System.out.print(num + " ");
    }

    checkNumbers(num+1, numbers, n);
}
}

```

22.5. First way - Без Метод – използване на асоциативен масив за Predicate<T> и за Consumer<T>

```

int n = Integer.parseInt(sc.nextLine());
List<Person> people = new ArrayList<>();

while (n-- > 0) {
    String[] tokens = sc.nextLine().split(", ");
    people.add(new Person(tokens[0], Integer.parseInt(tokens[1])));
}

Map<String, Predicate<Person>> predicateMap = new HashMap<>();

String ageCondition = sc.nextLine();
int age = Integer.parseInt(sc.nextLine());
predicateMap.put("younger", person -> person.getAge() <= age);
predicateMap.put("older", person -> person.getAge() >= age);

Map<String, Consumer<Person>> consumerMap = new HashMap<>();

consumerMap.put("name", person -> System.out.println(person.getName()));
consumerMap.put("age", person -> System.out.println(person.getAge()));
consumerMap.put("name age", person -> System.out.println(person.getName() + " - " + person.getAge()));

String format = sc.nextLine();

people.stream()
    .filter(predicateMap.get(ageCondition))
    .forEach(consumerMap.get(format));

```

22.6. Second way - Passing Functions to Method – Използване на асоциативен масив за Predicate<T> и за Consumer<T>

```
public class FilterByAge2ndWay {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int n = Integer.parseInt(sc.nextLine());
        LinkedHashMap<String, Integer> people = new LinkedHashMap<>();

        while (n-- > 0) {
            String[] tokens = sc.nextLine().split(", ");
            people.put(tokens[0], Integer.parseInt(tokens[1]));
        }
        String condition = sc.nextLine();
        int age = Integer.parseInt(sc.nextLine());
        String format = sc.nextLine();

        Predicate<Integer> tester = createTester(condition, age);
        Consumer<Map.Entry<String, Integer>> printer = createPrinter(format);
        printFilteredStudent(people, tester, printer);
    }

    static Predicate<Integer> createTester(String condition, Integer age) {
        Predicate<Integer> tester = null;
        switch (condition) {
            case "younger":
                tester = x -> x <= age;
                break;
            case "older":
                tester = x -> x >= age;
                break;
        }
        return tester;
    }

    static Consumer<Map.Entry<String, Integer>> createPrinter(String format) {
        Consumer<Map.Entry<String, Integer>> printer = null;
        switch (format) {
            case "name age":
                printer = person -> System.out.printf("%s - %d%n", person.getKey(),
person.getValue());
                break;
            case "name":
                printer = person -> System.out.printf("%s%n", person.getKey());
                break;
            case "age":
                printer = person -> System.out.printf("%d%n", person.getValue());
                break;
        }
        return printer;
    }

    static void printFilteredStudent(LinkedHashMap<String, Integer> people,
Predicate<Integer> tester, Consumer<Map.Entry<String, Integer>> printer) {

        for (Map.Entry<String, Integer> person : people.entrySet()) {
            if (tester.test(person.getValue()))
                printer.accept(person);
        }
    }
}
```

```
        }
    }
```

22.7. Какво се крие зад `.filter()` и зад `.foreach()` и зад `.removeIf()` при използването им в поток от данни

`.filter()`
`Stream<T> filter(Predicate<? super T> predicate);` - върни стрийм от тип `T` като използваш функция `Predicate` с входен параметър `T` и изход `true` или `false`

`.foreach()`

`void forEach(Consumer<? super T> action);` - върни при зададен параметър от тип `T` върни `void`

`.removeIf()`

`Predicate<Integer> ifDivisibleByN = x -> x % n != 0;`
`numbers.removeIf(ifDivisibleByN);`
`default boolean removeIf(Predicate<? super E> filter) {}` - ползва предикат

22.8. Lazy evaluation in JAVA

Няма да бъде изпълнена даден стрийм поток от команди (Stream API chaining), докато не се терминира потока

Пример 1:

```
IntStream.rangeClosed(lower, upper)
    .boxed()
    .filter(filter) - ако намери стойност, то я подава на следващата команда от chaining-a.  
Но ако не намери, то не я подава! По-оптимално е.  
    .forEach(printer); - тук forEach терминира chaining-a
```

Пример 2:

```
IntStream intStream = numbers
    .stream()
    .filter(x -> x % 2 == 0)
    .mapToInt(x -> x * 2); - до тук все още не се е изпълнило
```

`System.out.println(intStream.sum());` - тук вече се изпълнява стрийма, `.sum()` е терминираща операция

Пример 3: - в този код резултата на сумата ще е 0

```
int[] factor = new int[]{2};
IntStream intStream = numbers
    .stream()
    .filter(x -> x % 2 == 0)
    .mapToInt(x -> x * factor[0]); - не прави нищо до момента.
```

`factor[0] = 0;` - сменяме стойността на фактора
`System.out.println(intStream.sum());` - получаваме нула

OOP part

23. Working with Abstraction

Абстракция не може да се дефинира с конкретика. Точно затова конкретизираме създавайки повече класове и обекти.

23.1. Project Architecture

Splitting code into Methods – колкото може повече да правим на методи с подходящото име – и друг програмист като чете, да го разбира

Splitting code into Classes

Да не кръщаваме променливите/полетата като типа, от който са

`private Point bottomLeftPoint;` - грешно

`private Point bottomLeft;` - **вярно**

Projects

23.2. Code Refactoring

- **Restructures** code without changing the behaviour
- **Improves** code readability
- **Reduces** complexity

Refactoring techniques

- **Breaking code** into reusable units
- **Extracting** parts of methods and classes into new ones

`depositOrWithdraw()`

`deposit()`
`withdraw()`

Improving names of variables, methods, classes, etc.

`String str;`

`String name;`

Moving methods or fields to more appropriate classes

`Car.open()`

`Door.open()`

Започваме рефакториране или от най-малкото парче код или от най-голямото парче код

23.3. Enumerations

Като опростен Map работи enum

- Represent a numeric value from a fixed set as a text
- We can use them to pass **arguments** to **methods** without making code confusing
- By default **enums** start at 0
- Every next value is incremented by 1
- **Enumerations** define a fixed **set of constants/integers** – **статични static** константи
- Represent **numeric values/integers**
- We can easily **cast enums to numeric types**
- **We can re-use many times the enums**
- **Enum** се използва за **hard-core-нати** стойности
- **Достъпен до други класове е enum, докато примерно Map е достъпен само за текущия клас**

Пишем енумерациите с главни букви – като при конвенцията за изписване на класове
При enum, пишем първо стойностите на енумарацията (Spring, Summer, ..)

```

public enum CardSuit {
    CLUBS,
    DIAMONDS,
    HEARTS,
    SPADES;
}
for (CardSuit value : CardSuit.values()) {
    System.out.printf("Ordinal value: %d; Name value: %s%n", value.ordinal(), value.name());
}

value.ordinal() - връща поредния номер на даденото поле като започва от 0, т.е. начинът, по който
са подредени полетата

value.name() - връща името на даденото поле
същото като
value.toString() - за всяко енум поле връща името на даденото поле

```

Когато имаме (2), (4).., то трябва да използваме конструктор

```

package HotelReservation;
public enum Season {
    Spring(2), Summer(4), Autumn(1), Winter(3);

    private final int seasonIndex;

    Season(int index) {
        this.seasonIndex = index;
    }

    public int getSeasonIndex() {
        return this.seasonIndex;
    }
}

```

Season.Summer.name() - връща името на полето Summer(4), т.е. връща Summer
Season.Summer.getSeasonIndex() - връща 4

```

package HotelReservation;
enum Discount {
    None(0),
    SecondVisit(10),
    VIP(20);

    private final int discountPercents;

    Discount (int percents) {
        this.discountPercents = percents;
    }

    public double getDiscount() {
        return (100 - this.discountPercents) / 100.0;
    }
}

package HotelReservation;
import java.util.Scanner;

```

```

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        double pricePerDay = Double.parseDouble(sc.next());
        int numberOfDays = sc.nextInt();
        String seasonName = sc.next();
        String discountType = sc.next();

        Season season = Season.valueOf(seasonName); //връща съответното ПОЛЕ от енумерацията
        Discount discount = Discount.valueOf(discountType); //връща съответното ПОЛЕ от енумерацията
    }
}

private static double getPriceFor(double pricePerDay, int numberOfDays, Season season,
Discount discount) {
    return pricePerDay * season.getSeasonIndex() * numberOfDays * discount.getDiscount();
}
}

```

Друг пример

```

public enum TrafficLight {
    RED,
    GREEN,
    YELLOW
}
TrafficLight[] trafficLightsArrayToChange = Arrays.stream(sc.nextLine().split("\\s+"))
    .map(e -> TrafficLight.valueOf(e)) //mapping to Enum - за всеки текст, създава обект с
определеното enum Поле
    .toArray(TrafficLight[]:: new);

TrafficLight[] lightsEnum = TrafficLight.values(); // връща

```

```

trafficLightsArrayToChange = [TrafficLight@854 "GREEN"
    E 0 = {TrafficLight@854} "GREEN"
        > f name = "GREEN"
        > f ordinal = 1
    E 1 = {TrafficLight@855} "RED"
        > f name = "RED"
        > f ordinal = 0
    E 2 = {TrafficLight@856} "YELLOW"
        > f name = "YELLOW"
        > f ordinal = 2
]

```

litghsEnum[0] е Green
TrafficLight.values()[0]; е Green

Пример на енумерация с тип double

```
public enum ToppingType {
    MEAT(1.2),
    VEGGIES(0.8),
    CHEESE(1.1),
    SAUCE(0.9);

    private double modifier;

    ToppingType(double modifier) {
        this.modifier = modifier;
    }

    public double getModifier() {
        return this.modifier; //връща 1.2 или 0.8 и т.н.
    }
}
```

Друг пример за enumerations:

```
public enum Corps {
    AIRFORCE("Airforces"),
    MARINE("Marines");

    private final String name;

    Corps(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}

Corps.AIRFORCE || Corps.MARINE;
Corps.AIRFORCE.getName() || Corps.MARINE.getName()
```

Пример за енумерация като всяко поле има по няколко данни

```
public enum Season {
    SPRING("March", "April", "May"),
    SUMMER("June", "July", "August"),
    AUTUMN("September", "October", "November"),
    WINTER("December", "January", "February");

    private String m1;
    private String m2;
    private String m3;

    Season(String m1, String m2, String m3) {
        this.m1 = m1;
        this.m2 = m2;
        this.m3 = m3;
    }

    public String[] returnMonths(){
        String[] s = (this.m1 + " " + this.m2 + " " + this.m3).split(" ");
        return s;
    }
}
```

```

public class Main {
    public static void main(String[] args) {
        System.out.println(Season.AUTUMN);
        System.out.println(String.join(", ", Season.AUTUMN.returnMonths()));
    }
}

```

AUTUMN

September, October, November

23.4. Static Keyword in Java

- Used for **memory management** mainly
- Can apply with:
 - Nested class
 - Variables
 - Methods
 - Blocks
- Belongs to the class than to an instance of the class

Static Class

- A **top-level** class is a class that is not a nested class
- A **nested** class is any class whose declaration occurs within the body of another class or interface
- Only nested classes can be **static**

```

class TopClass {
    static class NestedStaticClass {
    }
}

```

Static Variable

- Can be used to refer to the **common** variable of all objects – например, един лихвен процент за депозити за всички клиенти на банката / всички обекти на класа
- Allocate memory only once in class area at the time of class loading

Static Method

- Belongs to the class rather than the object of a class
- Can be **invoked** without the need for creating an instance of a class
- Can **access** static data member and can **change** the value of it
- Can **not use non-static** data member or call **non-static method** directly
- **this** and **super** cannot be used in static context
- зарежда се първоначална памет само статичните данни и методи, и когато имаме думата new, се заделят памет допълнително.
- **Static** не знае за инстанции

```

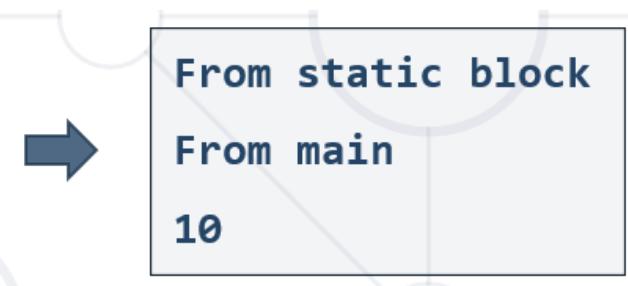
class Calculate {
    static int cube(int x) { return x * x * x; }
    public static void main(String args[]) {
        int result = Calculate.cube(5);
        System.out.println(result);           // 125
        System.out.println(Math.pow(2, 3));   // 8.0
    }
}

```

Static Block – изпълнява се първи от JVM, преди всичко друго

- A set of **statements**, which will be executed by the JVM **before execution** of **main** method
- Executing **static block** is at the time of class loading – изпълняват се при зареждане на метода
- A class can take any number of static block but all blocks will be executed **from top to bottom**
- Използват се, за да setup-нат някакви неща при стартирането на нашата програма

```
class Main {  
    static int n;  
    public static void main(String[] args) {  
        System.out.println("From main");  
        System.out.println(n);  
    }  
  
    static {  
        System.out.println("From static block 1");  
        n = 10;  
    }  
}
```



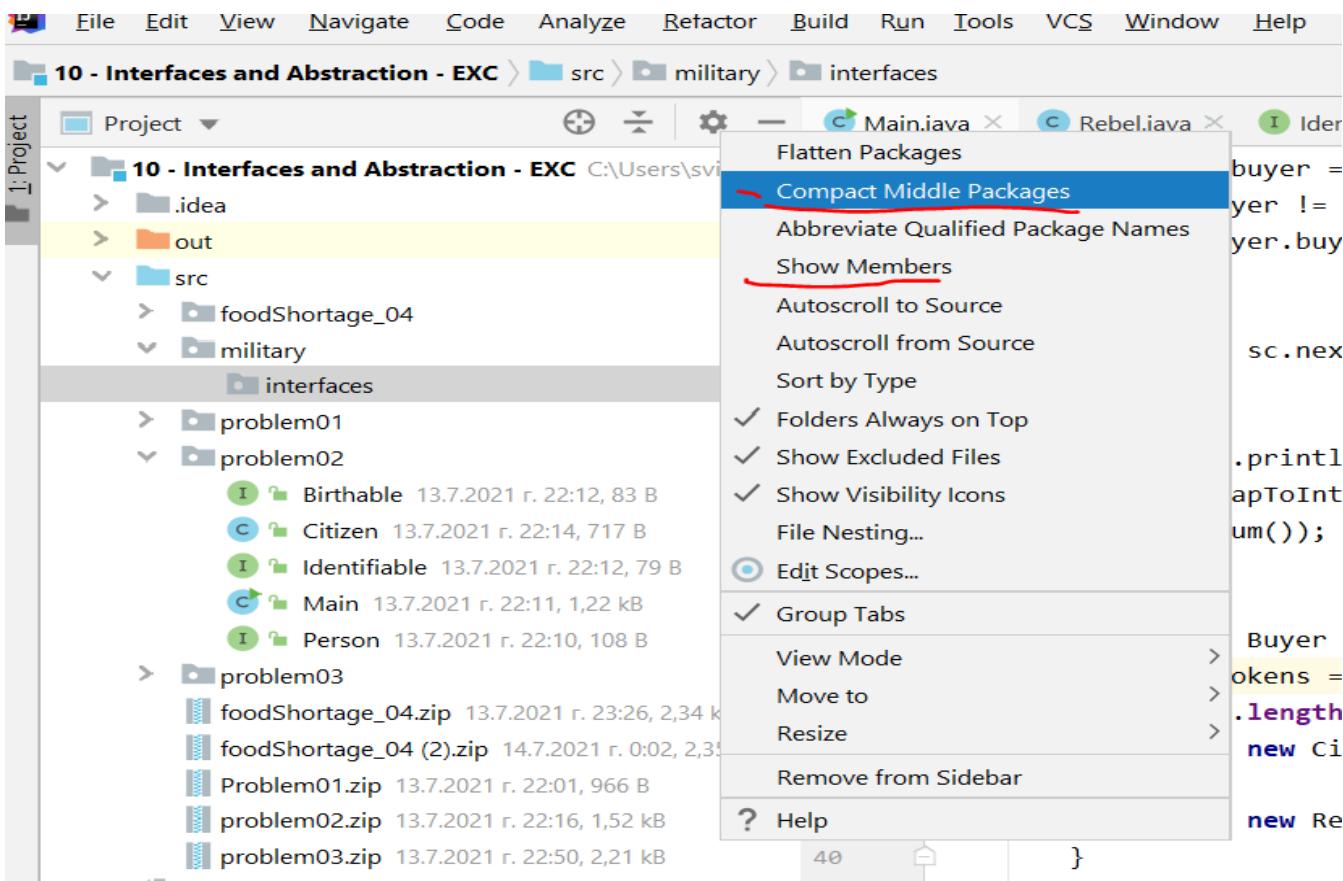
Статичните блокове се изпълняват един след друг спрямо поредността на кода ни

```
public class Main {  
    static {  
        System.out.println("Static block 1");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Main block");  
    }  
  
    static {  
        System.out.println("Static block 2");  
    }  
}
```

The diagram shows a vertical sequence of three grey boxes. From top to bottom, they contain the text: "Static block 1", "Static block 2", and "Main block". A grey arrow points from the first box down to the second, and another arrow points from the second down to the third.

23.5. Packages in Java

- Used to group related classes
- Like a folder in a file directory
- Use packages to avoid name conflicts and to write a better maintainable code
- Packages are divided into two categories:
 - **Built-in Packages** (packages from the **Java API**)
 - User-defined Packages (create own packages)



Build-In Packages

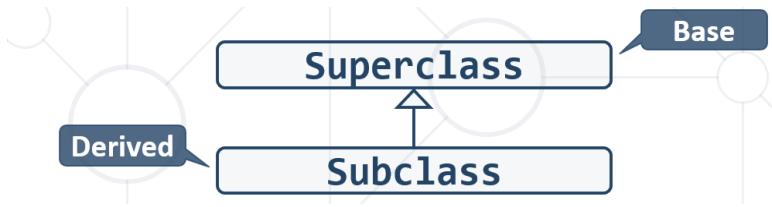
- The library is divided into packages and classes
- Import a single class or a whole package that contain all the classes
- To use a class or a package, use the import keyword
- The complete list can be found at Oracles website: <https://docs.oracle.com/en/java/javase/>

```
import package.name.Class; // Import a single class
import package.name.*;    // Import the whole package
```

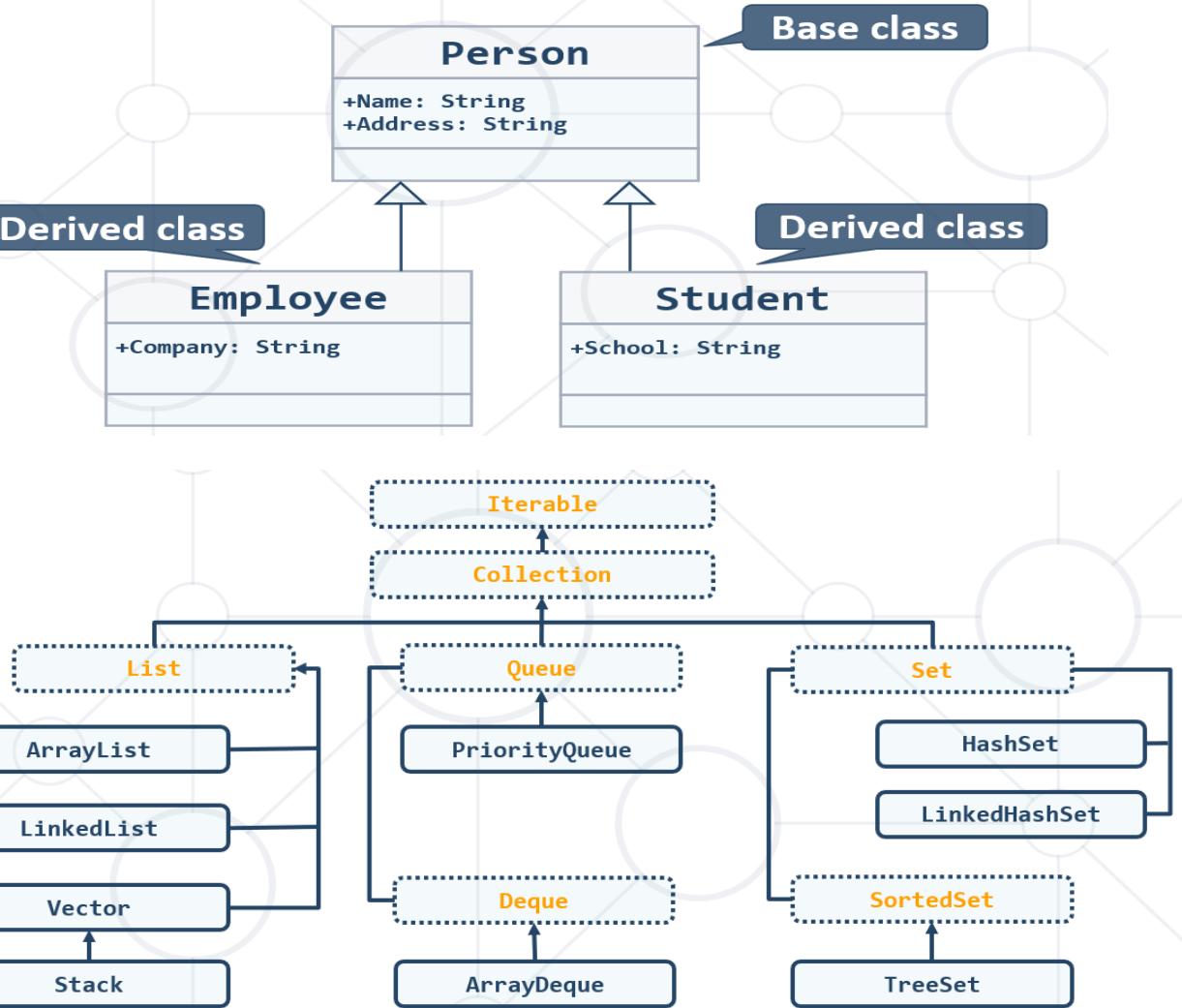
24. Inheritance – extending classes

24.1. Introduction

- Superclass - Parent class, Base Class**, the class giving its members to its child class, базовия клас не знае кой ги наследява
- Subclass - Child class, Derived Class**, The class taking members from its base class, детето клас използва полетата на базовия клас



- Inheritance leads to **hierarchies** of classes and/or interfaces in an application:



- **Object** is at the root of Java Class Hierarchy
- Java supports inheritance through **extends** keyword
- Class **taking all members** from another class

```
class Person { ... }
class Student extends Person { ... }
class Employee extends Person { ... }
```

- You can access inherited members

```
class Person { public void sleep() { ... } }
class Student extends Person { ... }
class Employee extends Person { ... }
```

```
Student student = new Student();
student.sleep();
Employee employee = new Employee();
employee.sleep();
```

- Reusing Constructors
 - Constructors are **not inherited**
 - Constructors **can be reused** by the child classes

```
class Student extends Person {
    private School school;
    public Student(String name, School school) {
```

```

super(name); // Constructor call should be first - извиква конструктора на базовия клас
this.school = school; //извиква поле на текущия клас
}
}

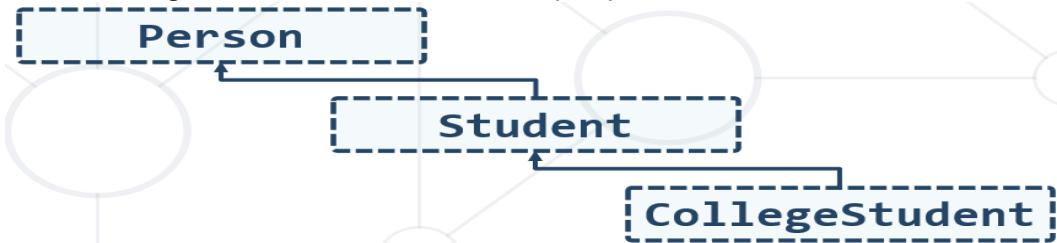
```

- Inheritance has a transitive relation

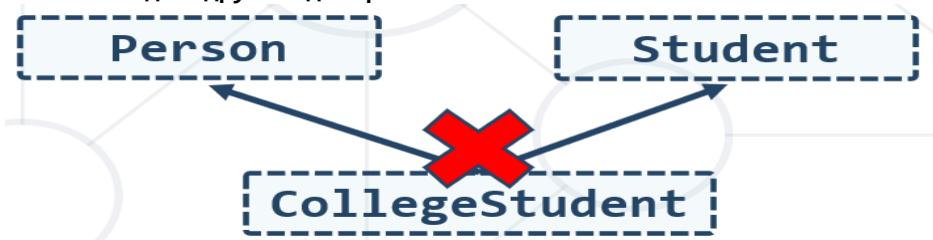
```

class Person { ... }
class Student extends Person { ... }
class CollegeStudent extends Student { ... }

```



- In Java there is no multiple inheritance for classes - няма как един клас да бъде наследник на два други едновременно



- Only multiple interfaces can be implemented

- Use the super keyword

```

class Person { ... }

class Employee extends Person {
    public void fire(String reasons) {
        System.out.println(super.name + //достъпваме полето name на базовия клас
            " got fired because " + reasons);
    }
}

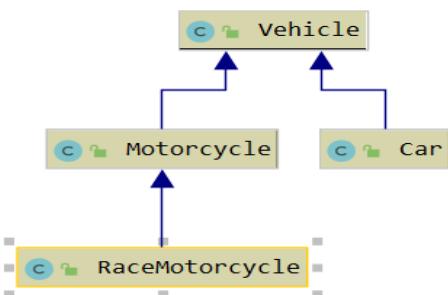
```

Super – в IntelliJ, като избереш super, то ти дава само валидните неща (които не са private)

Наследниците получават всичко, което не е private.

А в други класове използваме this. и super.

Наследник класа (Subclass - Child class, Derived Class) се интересува само от нещата на своя прям баща (Superclass - Parent class, Base Class) – т.e. думичката super реферира само едно ниво нагоре – НЕ Е съвсем така – ако в междуинния клас нямаме дефинирани други имплементации на даден метод, то например в класа RaceMotorCycle използваме super.setFuelConsumption(DEFAULT_FUEL_CONSUMPTION) като викаме метода от класа Vehicle като по този начин пропускаме класа Motorcycle!



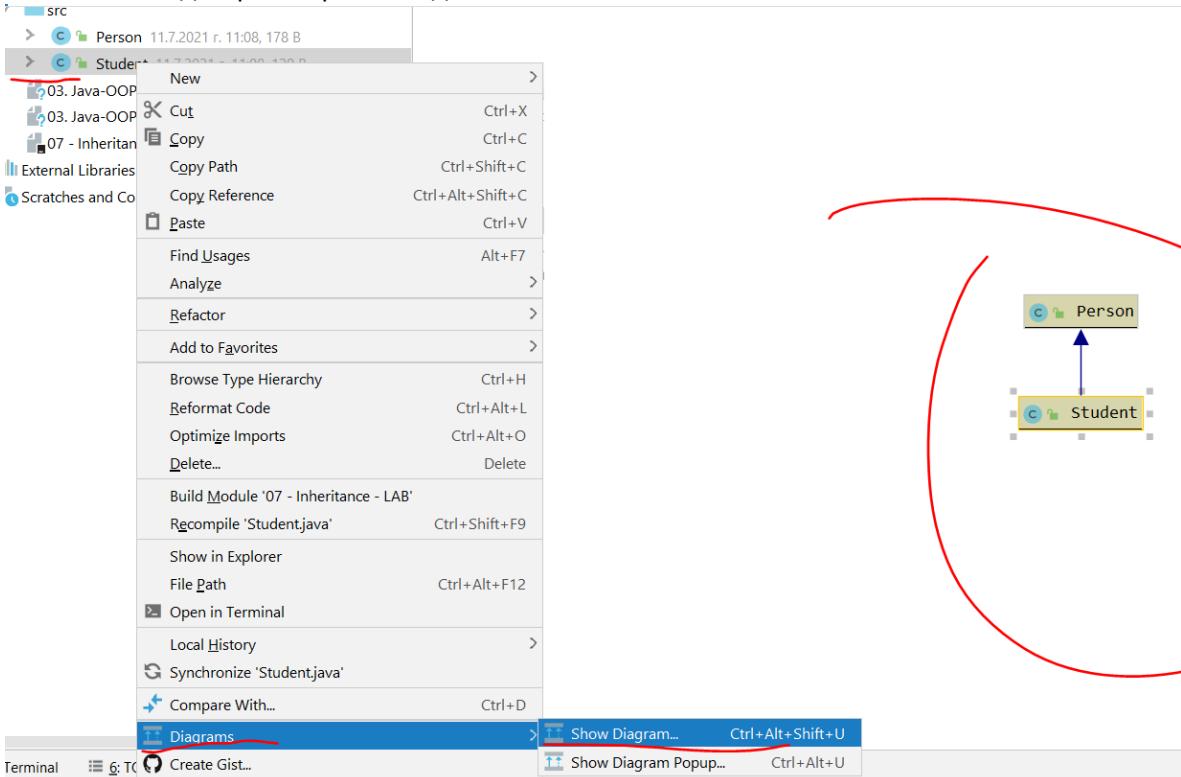
```

public class RaceMotorcycle extends Motorcycle {
    private final static double DEFAULT_FUEL_CONSUMPTION = 8.00;

    public RaceMotorcycle(double fuel, int horsePower) {
        super(fuel, horsePower);
        super.setFuelConsumption(DEFAULT_FUEL_CONSUMPTION);
    }
}

```

Показване на диаграма при наследяване



24.2. Reusing Classes

Inheritance and Access Modifiers

- Derived classes **can access all public and protected members**
- Derived classes can access **default members if in same package**
- Private fields aren't inherited** in subclasses (can't be accessed)

```

class Person {
    protected String address;
    public void sleep();
    private String id;
}

```

Shadowing Variables

- Derived classes **can hide** superclass variables

```
class Person { protected int weight; }
```

```

class Patient extends Person {
    protected float weight; //не работим с полето на класа Person, а с полето на класа Patient
    public void method() {
        double this.weight = 0.5d;
    }
}

```

```

    }
}



- Use super and this to specify member access



```

class Person { protected int weight; }

class Patient extends Person {
 protected float weight;

 public void method() {
 double weight = 0.5d; //Local variable
 this.weight = 0.6f; //Instance member
 super.weight = 1; //Base class member
 }
}

@Override
public String toString() {
 return super.toString(); //викаме toString метода на базовия клас
}

```


```

Overriding Derived Methods

- A child class can redefine existing methods, but method in base class must not be final

```

public class Person {
    public void sleep() {System.out.println("Person sleeping"); }
}

public class Student extends Person {
    @Override // презаписваме какво прави метода sleep в класа Student
    // Signature and return type should match:
    public void sleep(){ System.out.println("Student sleeping"); }
}

```

Final Methods

- `final` – defines a method that can't be overridden

```

public class Animal {
    public final void eat() { ... } // не може да се презаписва/променя
}

public class Dog extends Animal {
    @Override
    public void eat() {} // Error...
}

```

Final Constructors

There are no final constructors!!!!

Final Classes

Inheriting from a final classes is forbidden = a final class can not be extended!!!

```

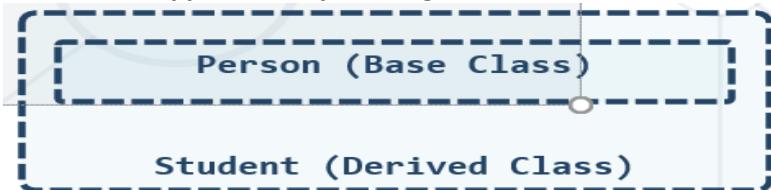
public final class Animal {
    ...
}

public class Dog extends Animal {} // Error...
public class MyString extends String {} // Error...
public class MyMath extends Math {} // Error...

```

Inheritance Benefits – Abstraction

- One approach for providing abstraction



```
Person person = new Person();
Student student = new Student();
```

Person student1 = new Student(); - по-високо ниво на абстракция, обектът student1 има методи само от класа Person, въпреки че в паметта се е създадо обект/инстанция от тип Student

```
List<Person> people = new ArrayList();

people.add(person);
people.add(student);
```

- We can extend a class that we can't otherwise change

We create CustomArrayList class to add random functionality of the class ArrayList

24.3. Types of Class Reuse

A) Extension = inheritance – by using the “extends”

- Duplicate code is error prone
- Reuse classes through extension
- Sometimes the only way



Example:

```
public class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
}

public class Reptile extends Animal {
    public Lizard(String name) {
        super(name);
    }
}

public class Lizard extends Reptile {
    public Lizard(String name) {
        super(name);
    }
}
```

B) Composition and delegation

- Using classes and fields to define classes in the composition

Монитора не е лаптоп, няма как да има унаследяване

```
class Laptop {  
    Monitor monitor;  
    Touchpad touchpad;  
    Keyboard keyboard;  
...  
}
```

Delegation – to delegate some job/implementation to the fields and classes in the class Laptop

```
class Laptop {  
    Monitor monitor;  
    void incrBrightness() {  
        this.monitor.brighten();  
    }  
  
    void decrBrightness() {  
        this.monitor.dim();  
    }  
}
```

24.4. When to Use Inheritance

IS-A relationship between classes – едното нещо дали е като другото нещо, например Cat и Dog са вид Animal

Derived class IS-A-SUBSTITUTE for the base class – дали можем да заместим с наследника базовия клас

Share the same role – имат същата роля

Derived class is the same as the base class but adds a little bit more functionality

24.5. Абстрактен метод и абстрактен клас

Казвайки на един клас, че е абстрактен, предупреждаваме компилатора, че този клас може да дефинира методи, чиято имплементация не се намира в текущия клас!

Тези методи без имплементация в текущия клас, са абстрактни методи.

```
public abstract class Car extends Vehicle {  
    ...  
    public abstract void reFuel(); //без тяло на метода, без имплементация, задължава  
    всички/поне един наследници да имплементират някаква логика за този метод!!!  
}  
  
public class FamilyCar extends Car {  
    ....  
    @Override  
    public void reFuel() {  
        ...  
    }  
}
```

При интерфейса имаме само поведение, само абстрактни методи/класове. Без никаква имплементация. При абстрактните класове и методи имаме както поведение на абстрактни методи, така и имплементация на обикновени неабстрактни методи.

25. Encapsulation

25.1. Hiding Implementation of data – Encapsulation

Abstraction solves the problem at design level and **encapsulation at implementation level**



- **Process of wrapping code and data together into a single unit** – обвиване на код и данни в единица
- Flexibility and extensibility of the code
- Reduces complexity – не трябва да се интересувам как толкова много нещата, важното е че мога да го използвам
- Structural changes remain local – ако променим нещо отвън, то няма как да промени енкапсулраната единица
- Allows validation and data binding

- Objects fields **must be private = locked**

```
class Person {  
    private int age;  
}
```

- Use **getters** and **setters** for data access = **unlocked**

```
class Person {  
    public int getAge()  
    public void setAge(int age)  
}
```

- Fields should be **private**
- **Accessors(getters)** and **Mutators(setters)** should be **public**

This

- **this** is a reference to the **current object** = **this** can refer to current class instance

```
public Person(String name) {  
    this.name = name;  
}  
  
▪ this can invoke current class method  
public String fullName() {  
    return this.getFirstName() + " " + this.getLastName();  
}
```

- `this` can invoke current class constructor

```
public Person(String name) {
    this.firstName = name;
}

public Person (String name, Integer age) {
    this(name); //инстанцията се създава
    this.age = age; //и тук инстанцията се създава!!! – ако се разменят двета реда, ще има грешка
}
```

25.2. Смисълът на енкапсулацията

Заради security причини.

Полетата трябва да бъдат `private`. Докато `getters` и `setters` могат да бъдат публични.

Точно в тези `getters` и `setters` можем:

- да сложим `if` проверка,
- да зададем някакъв критерий какво да връщаме
- дали винаги можем да сетнем/създадем обект
- при `getter`-а да върнем `unmodifiable list`

Т.е. за контрол и security моделиране.

25.3. Access Modifiers

`private` -> `Package Private(Default)` -> `protected` -> `public`

25.2.1. Private Access Modifier

- Object hides data from the outside world

```
class Person {
    private String name;
    Person (String name) {
        this.name = name;
    }
}
```

- Classes and interfaces **cannot** be private
- Data can be **accessed only within the declared class itself**

25.2.2. Default Access Modifier = Package Private

- **Do not explicitly declare** an access modifier

```
class Team {
    String getName() {...}
    void setName(String name) {...}
}
```

- **Available** to any other class in the same **package**

```
Team real = new Team("Real");
real.setName("Real Madrid");
System.out.println(real.getName());
```

25.2.3. Protected Access Modifier

- Grants **access to subclasses only no matter in which package** and to all other non-subclasses in the same package!!!

```
class Team {
    protected String getName () {...}
    protected void setName (String name) {...}
}
```

- **protected** modifier cannot be applied to classes and interfaces
- Prevents a **nonrelated** class from trying to use it

25.2.4. Public Access Modifier

- Grants access to **any class** belonging to the **Java Universe**

```
public class Team {
    public String getName() {...}
    public void setName(String name) {...}
}
```

- Import a package if you need to use a class
- The **main()** method of an application must be **public**

25.4. Validation

- **Data validation** happens in **setters**

В клас Person

```
private void setSalary(double salary) {
    if (salary < 460) {
        throw new IllegalArgumentException("Message");
    }

    this.salary = salary;
}
```

- Constructors use **private setters** with validation logic - Validation happens inside the setter
- Guarantees **valid state** of object in its creation
- Guarantees **valid state** for public setters

Конструктор в клас Person

```
public Person(String firstName, String lastName, int age, double salary) {
    this.setFirstName(firstName);
    this.setLastName(lastName);
    this.setAge(age);
    this.setSalary(salary);
}
```

В метода main

```
try {
    Person person = new Person(input[0], input[1], Integer.parseInt(input[2]),
Double.parseDouble(input[3]));
    people.add(person);
} catch (IllegalArgumentException exc){
    System.out.println(exc.getMessage());
}
```

25.5. Mutable and Immutable Objects

Objects

- Mutable Objects - The contents of that instance **can** be altered
- Immutable Objects - The contents of the instance **can't** be altered

```
String str = new String("old String");
System.out.println(str);
```

```
str.replaceAll("old", "new"); //old String  
System.out.println(str); // отново old String
```

Mutable Fields

- **private** mutable fields are not fully encapsulated
- In this case **getter is like setter too** – или като извикаме в Main-а листа, може да му добавяме веднага елементи, а ние искаме да му добавяме елементи само през името на класа Team

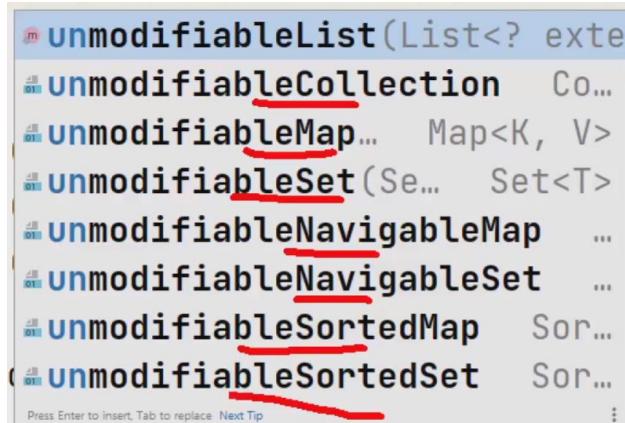
```
class Team {  
    private String name;  
    private List<Person> players;  
  
    public List<Person> getPlayers() {  
        return this.players;  
    }  
}
```

Immutable Fields

- For securing our collection we can return **Collections.unmodifiableList()**

```
class Team {  
    private List<Person> players;  
  
    public void addPlayer(Person person) {  
        this.players.add(person); // Add new methods for functionality over list  
    }  
  
    public List<Person> getPlayers() {  
        return Collections.unmodifiableList(players); // Returns a safe collections – на която не можем да добавяме  
        // елементи, не можем да съзваме елементи, не можем да изтриваме елементи  
        Collections.unmodifiableCollection(models);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Team team = new Team("Loko");  
        Person playerOne = new Person("Pesho", "Goshov", 22);  
        team.getFirstTeam().add(playerOne); // дава грешка java.lang.UnsupportedOperationException
```

Можем да използваме



25.6. Keyword Final

- **final class** can't be extended

```
public class Animal {}  
public final class Mammal extends Animal {}  
public class Cat extends Mammal {} // не може да екстендуваме Mammal!!!
```

- **final method** can't be overridden

```
public final void move(Point point) {}  
  
public class Mammal extends Animal {  
    @Override  
    public void move() {} // не може да го override-ваме в тялото на наследника!!!  
}
```

- **final variable** value can't be changed once it is set

```
private final String name;  
private final List<Person> firstTeam;  
public Team (String name) {  
    this.name = name;  
    this.firstTeam = new ArrayList<Person> ();  
}  
public void doSomething(Person person) {  
    this.name = ""; // дава грешка  
    this.firstTeam = new ArrayList<>(); // дава грешка  
    this.firstTeam.add(person); // ок е  
}
```

- **final constructor** – we can not have final constructor

26. Interfaces and Abstraction

26.1. Abstraction in OOP

- **Abstraction** means ignoring **irrelevant** features, properties, or functions and emphasizing the **relevant ones** ...
- ... **relevant** to the **context** of the **project** we develop
- Abstraction helps **managing** complexity
- Abstraction lets you focus on **what the** object does instead of **how it does it**

Да видим абстракция значи да разделим повече кода на нива.

Achieving Abstraction - there are 2 ways to achieve abstraction in Java:

- Interfaces (**100% abstraction**)
- Abstract class (**0% - 100% abstraction**)

```
public interface Animal {}  
public abstract class Mammal {}  
public class Person extends Mammal implements Animal {}
```

Abstraction

- Process of **hiding** the **implementation details** and showing only functionality to the user – какво прави без детайлите
- Achieved with **interfaces** and **abstract classes**
- **Abstraction can not be instantiated**

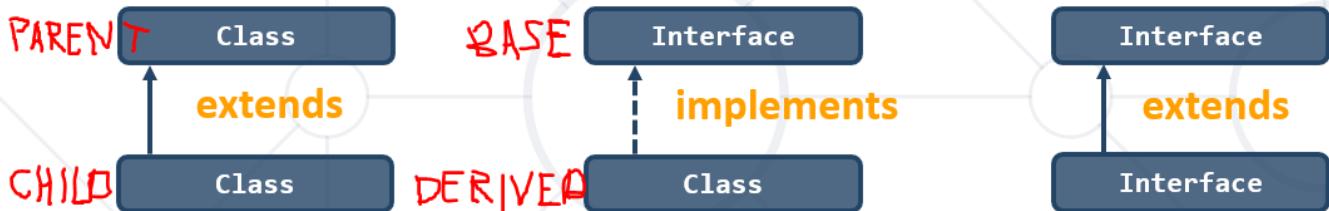
Encapsulation

- Process of wrapping code and data together into a single unit – обвиване на код и данни в единица
- Used to **hide the code** and **data** inside a **single unit** to **protect** the data from the outside world – как го правя с детайли
- Achieved with **access modifiers** (private, protected, public, default)

Abstraction solves the problem at design level and **encapsulation at implementation level**

26.2. Interface

▪ Relationship between classes and interfaces



▪ Multiple inheritance



Interface Example

Implementation of **print()** is provided in class **Document**

```
public interface Printable {  
    public static final int SOMENUMBER; //всички полета на интерфейс са public static final  
  
    public abstract void print(); //в повечето случаи всички методи в interface са public и abstract  
}  
  
class Document implements Printable {  
    public void print() { System.out.println("Hello"); } //implementing  
    public static void main(String args[]) {  
        Printable doc = new Document(); //Polymorphism - ДА  
        doc.print(); // Hello  
    }  
}
```

- Interface can **extend another interface**

```
public interface Showable {  
    void show();  
}
```

```
public interface Printable extends Showable {  
    void print();  
}
```

- Class which implements **child** interface must provide implementation for **parent** interface too

```
class Circle implements Printable  
public void print() {  
    System.out.println("Hello");  
}  
public void show() {  
    System.out.println("Welcome");  
}
```

Важно

Отляво е същият или по-базовия клас/интерфейс

Отляво е референцията, която да сочи към обекта в (рам) паметта

```
Seat seat = new Seat("Leon", "Gray", 110, "Spain", 11111.1); // отляво обектът има достъп до  
методите на Seat
```

```
CarImpl seat = new Seat("Leon", "Gray", 110, "Spain", 11111.1); // отляво обектът има достъп до  
методите на CarImpl
```

Понякога, за да направим колекция от различни обекти с общ ключов интерфейс референция (отляво), то може да се наложи да extend-нем интерфейсите, и да имаме базов интерфейс като общо начало/ключ. С чито методи на базовия интерфейс да работим.

Другият вариант е ако си направим абстрактен клас, който да extend-ва други абстрактни класове. За да можем да ползваме методите на базовия интерфейс, с който работим.

Default implementation in interface - Since Java 8 we can have method body in the interface –

```
public interface Drawable {  
    void draw();  
    default void msg() {  
        System.out.println("default method:");  
    }  
}
```

- If you need to override default method think about your **design – you should not do it!!!**
- Implementation is **not needed for default methods**
- **If you try to implement a default method, then the method should not be default!!!**
- Дафault-ния метод неискаме да го override-ваме в наследниците класове

Since Java 11, we can have static method in interface

```
public interface Drawable {  
    void draw();  
    static int cube(int x) { return x*x*x; }  
}  
  
public static void main (String args[]){  
    Drawable d = new Rectangle();  
    d.draw();  
    System.out.println(Drawable(cube(3));  
} // 27
```

Навигира от интерфейс метода към имплементацията на метода в класа наследник



```
@Override  
public int getHorsePower() {  
    return this.horsePower;  
}
```

Пример за използване на поле на интерфейс

```
public interface Car extends Serializable {  
    public final static int TYRES = 4;  
  
    String getModel();  
    String getColor();  
    Integer getHorsePower();  
    String countryProduced();  
}  
  
public class Seat implements Car {  
    private String model;  
    private String color;  
    private Integer horsePower;  
    private String countryProduced;  
  
    @Override  
    public String toString() {  
        return String.format("This is %s produced in %s and have %d tires", this.model,  
this.countryProduced,  
TYRES);  
    }  
}
```

26.2.1. record since Java 15 – a kind of data holder

The **record** is a new type of class in **Java** that makes it easy to create immutable data objects.

```
public class Point {  
    public final int x;  
    public final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}  
  
public record Point(int x,int y){  
}
```

26.3. Abstract Classes and methods

Abstract Classes

Cannot be instantiated

- May contain **abstract methods**

- Must provide **implementation** for all **inherited** interface members
- Implementing an interface might map the interface methods onto **abstract** methods

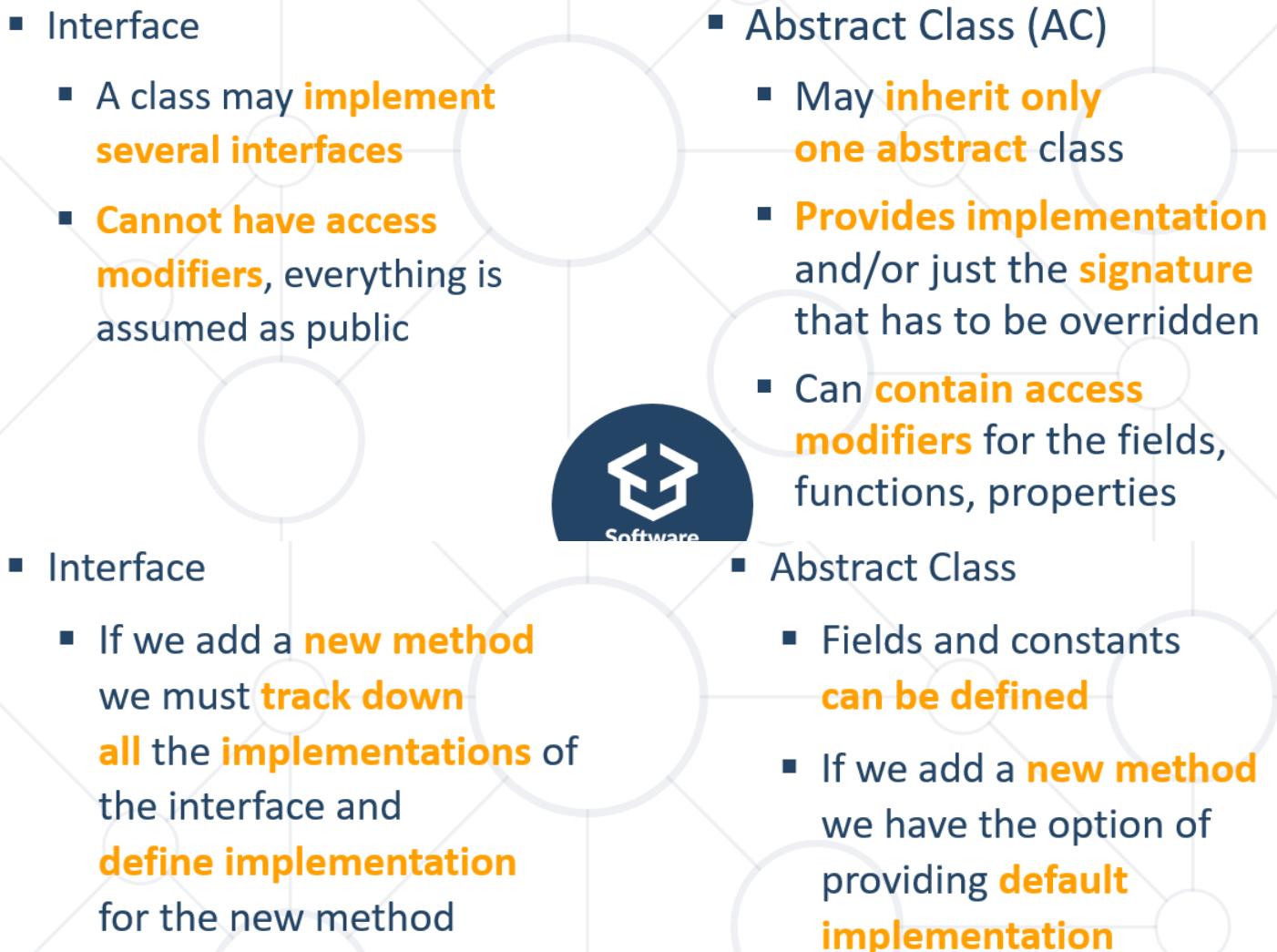
```
public abstract class Animal {  
}
```

Abstract Methods

- Declarations are only permitted in **abstract classes**
- Bodies must be **empty** (no curly braces)
- An abstract method declaration provides **no** actual implementation:

```
public abstract void build();  
protected abstract void build(); - абстрактният метод задължава наследника да имплементира
```

26.4. Interfaces vs Abstract Classes



26.5. More examples when working with interfaces

```
public interface Person {  
    public String getName();  
    public int getAge();  
}  
  
public class Citizen implements Person {
```

```

private String name;
private int age;

public Citizen(String name, int age) {
    this.name = name;
    this.age = age;
}

public String sayHello(){
    return "Hello";
}

@Override
public String getName() {
    return this.name;
}

@Override
public int getAge() {
    return this.age;
}
}

```

Reflection examples:

```

public static void main(String[] args) {
    Class[] citizenInterfaces = Citizen.class.getInterfaces(); // взема всички интерфейси, които
    класът Citizen имплементира – в случая взема интерфейса Person само.
    if(Arrays.asList(citizenInterfaces).contains(Person.class)){ //дали Person интерфейса се съдържа в
    citizenInterfaces
        Method[] fields = Person.class.getDeclaredMethods(); // връща методите на интерфейса

    Person person = new Citizen(name,age);
    System.out.println(person.sayHello()); //дава грешка, нямаме достъп до метода sayHello от класа
    Citizen

    Citizen citizen = new Citizen(name,age);
    System.out.println(citizen.sayHello()); // така работи

    Person person = new Citizen(name,age);
    System.out.println(((Citizen) person).sayHello()); //може да го кастнем и пак ще работи

    System.out.println(fields[0].getReturnType().getSimpleName()); //изкарва типа данни, който метода
    връща

```

27. Polymorphism

27.1. What is Polymorphism

Polymorphos = many shapes

Една и съща сигнатура/Reference Type, но различно поведение/Object Type.

В една и съща абстракция/референция, мога да разпиша различни форми(различни имплементации)

Референцията казва до кои неща/методи ще имаме достъп

- Such as a word having several different meanings based on the context
- Often referred to as the third pillar of OOP, after encapsulation and inheritance
- Ability of an object to take on many forms

Reference Type and Object Type

```

public class Person extends Mammal implements Animal {}
Animal person = new Person();
Mammal personOne = new Person();
Person personTwo = new Person();

```

Reference Type

Object Type

- **Variables** are saved in **reference type**
- You can use only **reference methods**
- If you need **object method**, you need to **cast it or override it**

- Check if **object** is an **instance** of a specific **class** – **да не го използваме/бави – част от Reflection е**

```
Mammal george = new Person();
Person peter = new Person();
```

Вариант 1

Check object type of person:

```
if (george instanceof Person) {} // да не го използваме/бави – част от Reflection е
```

Вариант 2

Cast to object type and use its methods

```
if (peter.getClass() == Person.class) {
    ((Person) peter).getSalary();
}
```

Горното не работи при Generics – generics се използва само докато кодът се компилира. Не можем да кажем george instanceof T

27.2. InstanceOf и полиморфизъм

```

public abstract class Person {
    protected String firstName;
    protected String lastName;

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}

public class Student extends Person{
    private String studentNumber;

    public Student(String studentNumber, String firstName, String secondName) {
        this.studentNumber = studentNumber;
        super.firstName = firstName;
        super.setLastName(secondName);
    }
}

```

```

ИЛИ така
public interface Person {
    public String getFirstName();
    public String getLastName();
}

public class Student implements Person{
    private String studentNumber;
    private String firstName;
    private String lastName;

    public Student(String studentNumber, String firstName, String secondName) {
        this.studentNumber = studentNumber;
        this.firstName = firstName;
        this.lastName = secondName;
    }

    @Override
    public String getFirstName() {
        return this.firstName;
    }

    @Override
    public String getLastName() {
        return this.lastName;
    }
}

```

```

public class Main {
    public static void main(String[] args) {
        Student st1 = new Student("1234", "Svilen", "Velikov");
        System.out.println(st1 instanceof Student); //true
        System.out.println(st1 instanceof Person); //true

        Person st2 = new Student("1234", "Svilen", "Velikov");
        System.out.println(st2 instanceof Student); //true
        System.out.println(st2 instanceof Person); //true
    }
}

```

27.3. Types of Polymorphism

27.3.1. Runtime Dynamic polymorphism (overriding) – we use here method overriding

```

public class Shape {}
public class Circle extends Shape {}
public static void main(String[] args) {
    Shape shape = new Circle();
}

```

- Also known as **Dynamic Polymorphism** - Using of **override** method

```

public class Rectangle {
    public Double area() {
        return this.a * this.b;
    }
}

```

```

public class Square extends Rectangle {
    @Override
    public Double area() {
        return this.a * this.a;
    }
}

public static void main(String[] args) {
    Rectangle rect = new Rectangle(3.0, 4.0);
    Rectangle square = new Square(4.0);

    System.out.println(rect.area());
    System.out.println(square.area()); //method overriding
}

```

Rules for Overriding Method

- Overriding can take place in **sub-class**
- Argument list must be the **same** as that of the **parent method**
- The overriding method must have **same return type**
- **Access modifier** cannot be more **restrictive**
- **Private, static** and **final** methods can **NOT** be overridden
- The overriding method **must not** throw new or broader **checked exceptions**

27.3.2. Compile time Static polymorphism (overloading) – we use here **method overloading**

```

int sum(int a, int b, int c){}
double sum(Double a, Double b){}

```

- Also known as **Static Polymorphism** - Argument lists could **differ** in:
 - Number of parameters
 - Data type of parameters
 - Sequence of Data type of parameters

```

static int myMethod(int a, int b) {}
static Double myMethod(Double a, Double b) {}

```

Rules for Overloading Method

- Overloading can take place in the **same class** or in its **sub-class**
- **Constructors** in Java can be **overloaded**
- Overloaded methods must have a **different argument list**
- Overloaded method should always be the part of the same class (can also take place in sub class), with **same name**, but **different parameters**
- They may have the **same** or **different return types**

27.4. Abstract Classes

- Abstract class **can NOT be instantiated**

```

public abstract class Shape {}
public class Circle extends Shape {}
Shape shape = new Shape(); // Compile time error
Shape circle = new Circle(); // polymorphism

```

- An **abstract class** may or may not include **abstract methods**
- If it has at least one abstract method, it must be declared **abstract**
- To use an **abstract class**, you need to **inherit it** afterwards

27.5. Постигане на полиморфизъм като референцията има достъп до исканите методи за всеки тип обект

Използваме setters в базовия абстрактен клас и прилагаме override за Rectangle, за Circle, и т.н.

```
public class Main {
    public static void main(String[] args) {
        Shape rect = new Rectangle(13D, 2D);
        System.out.println(rect.getPerimeter());
        System.out.println(rect.getArea());

        Shape circle = new Circle(3.02);
        System.out.println(circle.getPerimeter());
        System.out.println(circle.getArea());
    }

    public abstract class Shape {
        private Double perimeter;

        protected abstract void calculatePerimeter();

        protected void setPerimeter(Double perimeter){
            this.perimeter = perimeter;
        }

        public Double getPerimeter() {
            return this.perimeter;
        }
    }

    public class Rectangle extends Shape {
        private Double height;
        private Double width;

        public Rectangle(Double height, Double width) {
            this.height = height;
            this.width = width;
            this.calculatePerimeter();
        }

        @Override
        public void calculatePerimeter() {
            Double result = this.height * 2 + this.width * 2;
            super.setPerimeter(result);
        }
    }

    public class Circle extends Shape {
        private Double radius;

        public Circle(Double radius) {
            this.radius = radius;
            this.calculatePerimeter();
        }

        @Override
        protected void calculatePerimeter() {
            Double result = 2 * Math.PI * this.radius;
            super.setPerimeter(result);
        }
    }
}
```

27.6. Подходи при решаване на задачи

Когато не знаем дали ни трябва абстрактен клас или интерфейс, то винаги **започваме с интерфейс**.

Другият подход е **да започнем от класа**.

Никога не започваме от средата от **abstract class!!!**

Прилагането на интерфейс заедно с абстрактен клас е по-трудно.

Да избягваме type casting!!!

Когато имаме две референции към един и същи обект:

```
public class Bus extends Vehicle {}
```

```
main{
Map<String, Vehicle> vehicles = new LinkedHashMap<>();
Bus bus = new Bus(Double.parseDouble(tokens[1]), Double.parseDouble(tokens[2]),
Double.parseDouble(tokens[3]));
Vehicle bus2 = bus;
vehicles.put(tokens[0], bus);
}
```

27.7. Основни принципи в ООП

Is -> a дали е даден вид

Has -> a дали има някакво поле/свойство/метод в самия клас

Uses -> a дали използва друг обект / Strategy pattern

Важно - Полиморфизъм

1. Interface Vehicle

2. Abstract class VehicleImpl implements Vehicle

3. Class Bus extends VehicleImpl; Class Car extends VehicleImpl; Class Truck extends VehicleImpl

В main метода на Main класа използваме:

```
Vehicle bus = new Bus();
```

```
Vehicle car = new Car();
```

```
Vehicle truck = new Truck();
```

Object Slicing

Vehicle bus = new Bus(); - променливата bus няма достъп до методите на Bus, понеже имаме само достъп до методите на базовия интерфейс/клас Vehicle

28. SOLID = S.O.L.I.D.

Clean Architecture book: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series) 1st Edition

28.1. **S – Single responsibility principle** – class should only have one responsibility

- A class should **have only one responsibility**
 - Reduces **dependency** complexity
 - Each additional responsibility is an **axis to change the class**

```
public class HeroSettings {
    public static void changeName(Hero hero) {
```

```

        // Grant option to change
    }
}



- Still classes can have multiple methods
  - Each method should have single functionality, part of the class responsibility



```

public class HeroSettings {
 public static void changeName(Hero hero) {
 // Grant option to change name
 }
 public static void selectRole(Hero hero) {
 // Grant option to select role
 }
}

```


```

28.2. O - [Open–closed principle](#) – open for extension, but closed for modification

- Software entities (classes, modules, functions, etc.) should be
 - **open for extension** – чрез наследяване
 - **closed for modification** – добавянето на нова функционалност не променя старата функционалност – прилагаме Design patterns,
- **Design** the code in a way that **new** functionality can be added with **minimum changes** in the **existing** code

Extensibility

- Implementation takes future growth into consideration – да можем да погледнем в бъдещето и какво ще стане ако трябва да добавя нова функционалност на модула
- New or modified functionality affects little or not at all the internal structure and data flow of the system

Reusability

- Software reusability refers to **design features** of a software element that enhance its **suitability for reuse**
- Modularity
- **Low coupling** – х сочи към т, и у сочи към т
- **High cohesion** – нещата имат общо едно с друго/нещата които се променят по една и съща причина, трябва да бъдат на едно и също място или да са свързани
- [Coupling and Cohesion](#)

OCP – Violations

- **Cascading changes** through modules
- Each change **requires re-testing**
- Logic **depends** on conditional statements

OCP – Solutions

- Inheritance / Abstraction
- Inheritance / Template Method pattern
- Composition / Strategy patterns

```

public interface InitializableService {
    void init();
}

@Service
public class RimService implements InitializableService {

```

```

private final RimRepository rimRepository;

public RimService(RimRepository rimRepository) {
    this.rimRepository = rimRepository;
}

@Override
public void init() {
    if (rimRepository.count() == 0) {
        initRim("aluminium", "15");
        initRim("steel", "14");
        initRim("aluminium", "17");
    }
}

@Component
public class AppInit implements CommandLineRunner {
    private final RestTemplate restTemplate; //нашия Bean

    //Всички service класове, които сме имплементирали с InitializableService interface,
    //тук ни се зареждат автоматично
    private final List<InitializableService> allServices;

    public AppInit(RestTemplate restTemplate, List<InitializableService> allServices) {
        this.restTemplate = restTemplate;
        this.allServices = allServices;
    }

    @PostConstruct
    public void beginInit() {
        this.allServices.forEach(srvc -> srvc.init());
    }
}

```

28.3. L – [Liskov substitution principle](#) – objects should be replaceable with instances of their subtypes without altering the correctness of that program

What is Liskov Substitution?

- Качествено изпълняване наследяване на кода
- Derived types must be **completely substitutable** for their base types – **наследниците могат да заместват базовия клас**
- Reference to the base class can be replaced with a derived class without affecting the functionality of the program module – **използване на базовия клас или интерфейс**
- Derived classes extend without replacing the functionality of old classes – **наследници на базовия клас/интерфейс ги носим и в класовете на базовия клас/интерфейс**, като не им променяме функционалността

LSP Relationship

- **OOP Inheritance** : Student IS-A Person
- **Plus LSP** : Student IS-SUBSTITUTED-FOR Person

```

public static class Rectangle {
    int width;
    int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }
    public int getArea() {
        return this.width * this.height;
    }
}

public static class Square extends Rectangle {
    public Square(int side) {
        super(side, side);
    }
    @Override
    public int getArea() {
        return this.width * this.height; // това е грешно използване/дава друг резултат!!!
    }
}

```

OCP vs LSP

- Liskov Substitution Principle is just an **extension** of the Open Closed Principle and Single Responsibility principle
- We must make sure that new derived classes are extending the base classes **without changing** their **behavior** -

LSP – Violations and Solutions

- Violations
 - Type Checking – ако използваме **instance of** – не е **ок** ако проверяваме инстанцията на клас **наследник от кой тип е** – винаги трябва да е от типа на базовия клас
 - Overridden methods say "I am not implemented" – **оверрайдваме методи, които не правят нищо**
 - Base class depends on its subtypes
- Solutions
 - Refactoring in the **base class**

28.4. I – [Interface Segregation principle](#) – many specific interfaces are better than one general interface

ISP – Interface Segregation Principle

- Clients should **not be forced to depend** on methods they do **not use**
- Segregate interfaces
 - Prefer **small, cohesive** interfaces – **или interface Single Responsibility**
 - Divide "**fat**" interfaces into "**role**" interfaces

Fat interfaces

Classes whose **interfaces** are **not cohesive** have "fat" interfaces

```

public interface Worker {
    void work();
    void sleep();
}

public class Employee extends Worker{
    public void work() {Do something} //ok e
    public void sleep(Do something) //ok e
}

```

```

public class Robot implements Worker {
    public void work() {Do something} //ok е
    public void sleep() {
        throw new UnsupportedOperationException(); // не е ok
    }
}

```

Having "fat" interfaces:

- Classes have methods they do not use
- Increased **coupling – обвързаност увеличеност**
- Reduced flexibility
- Reduced maintainability

- Solutions to broken ISP
 - **Small** interfaces
 - **Cohesive** interfaces
 - Let the client **define** interfaces – "role" interfaces

Small and Cohesive "Role" Interfaces

```

public interface Worker {
    void work();
}

public interface Sleeper {
    void sleep();
}

public class Robot implements Worker {
    void work() {
        // Do some work...
    }
}

public class Employee implements Worker, Sleeper {
    void work() {
        // Do some work...
    }

    void sleep() {
        // Do sleep...
    }
}

```

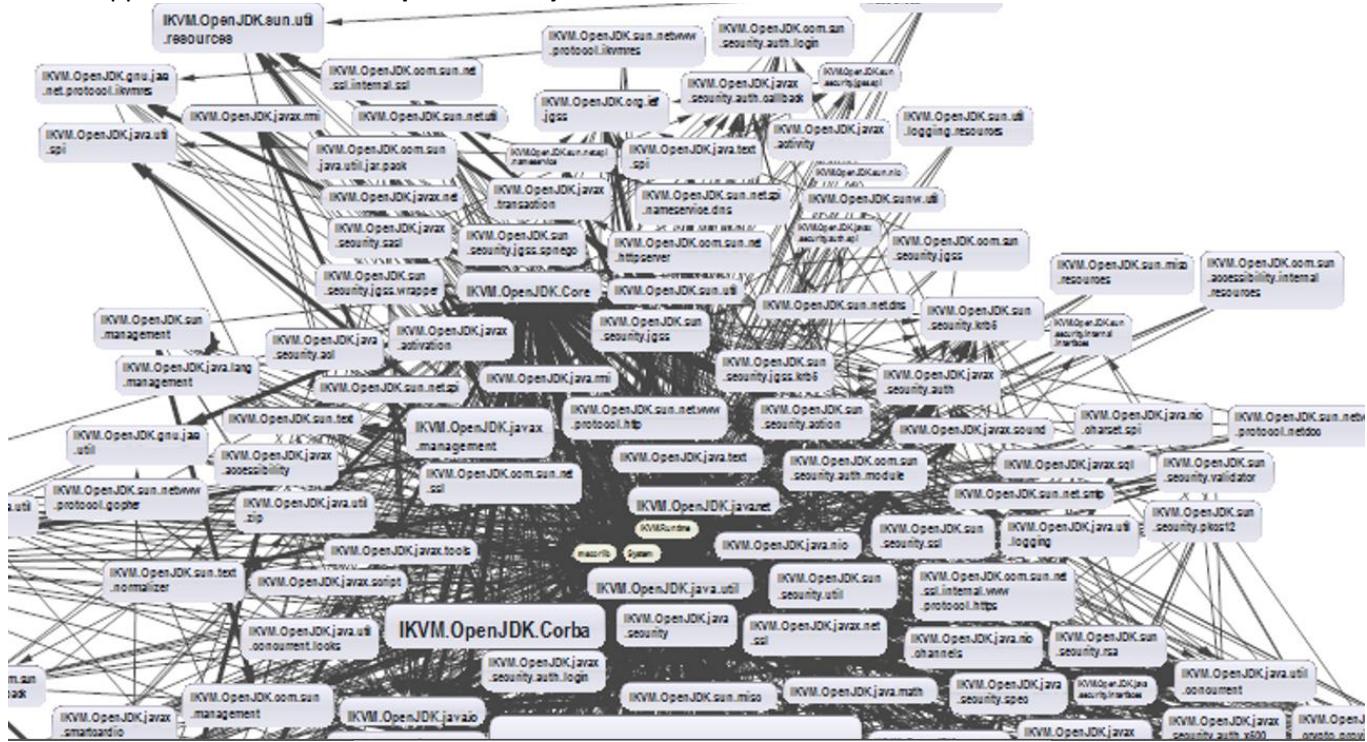
28.5. D – [Dependency inversion principle](#) – one should depend upon abstractions, not concretions

Dependency Inversion Principle (DIP) – one kind of implementation of the Inversion of Control principle

- **High-level modules should not depend on low-level modules** - both should **depend on abstractions**
- Abstractions should not depend on details / on implementation
- Details should depend on abstractions
- Goal: **decoupling between modules** through abstractions – чрез полиморфизъм, мога да инжектирам различно поведение, като методите да връщат по-абстрактния тип данни

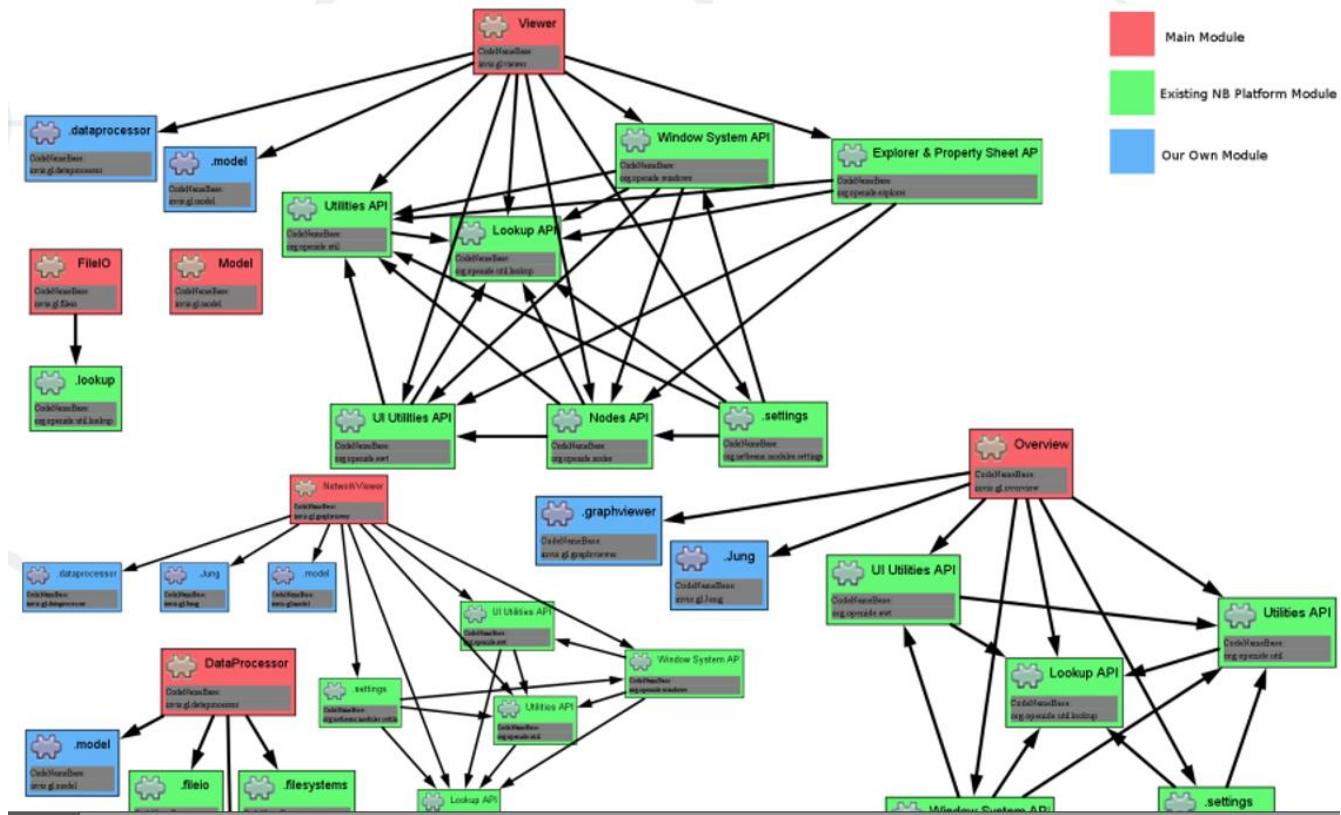
Ако в един interface има вложен клас, то това е в разрез с dependency inversion принципа.

What happens when modules depend directly on other modules – it becomes a mess



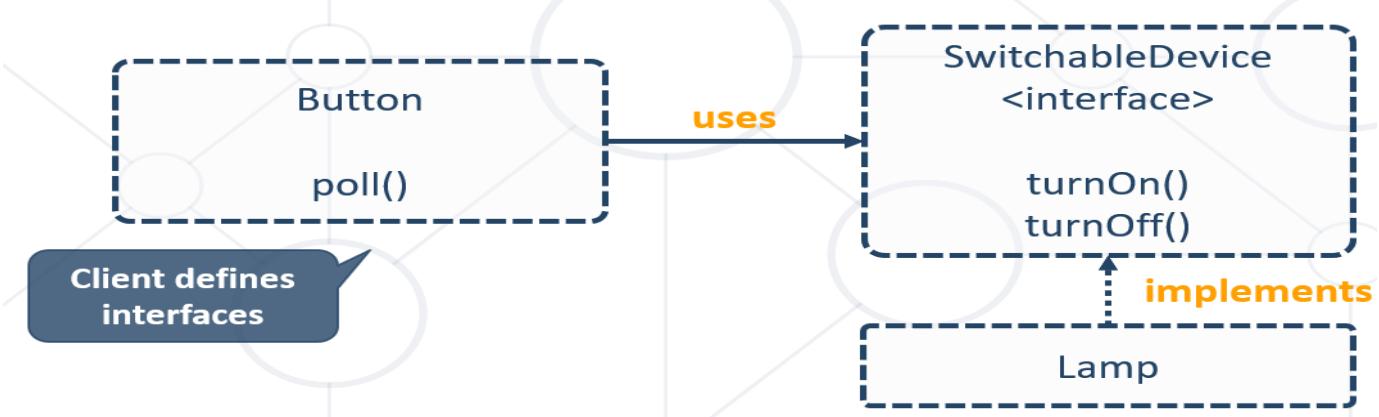
The goal is to **depend on abstractions**

Main module е абстракцията/interface примерно



Това, което е включва и изключва може да е лампа, кафе машина, микровълнова, кана за кафе:

- Find the abstraction independent of details



How to DIP? (1) – via the constructor – the best option!!! – инжектиране през конструктора

- Constructor injection - dependencies are passed through constructors

- Pros
 - Classes **self-documenting** requirements
 - Works well without container
 - Always **valid state**
- Cons
 - Many parameters
 - Some methods may not need everything

```

public class Copy {
    private Reader reader;
    private Writer writer;
    public Copy(Reader reader, Writer writer) {
        this.reader = reader;
        this.writer = writer;
    }
    public void copyAll() {}
}
  
```

How to DIP? (2) – via the setters

- Setter Injection - dependencies are passed through setters

- Pros
 - Can be changed anytime
 - Very **flexible**
- Cons
 - Possible **invalid state** of the object – може да не е създаден обекта writer примерно
 - Less intuitive

```

public class Copy {
    private Reader reader;
    private Writer writer;
    public void setReader(Reader reader) { this.reader = reader; }
    public void setWriter(Writer writer) { this.writer = writer; }
    public void copyAll() {}
}
  
```

How to DIP? (3) – via the method parameters

Parameter injection - dependencies are passed through method parameters

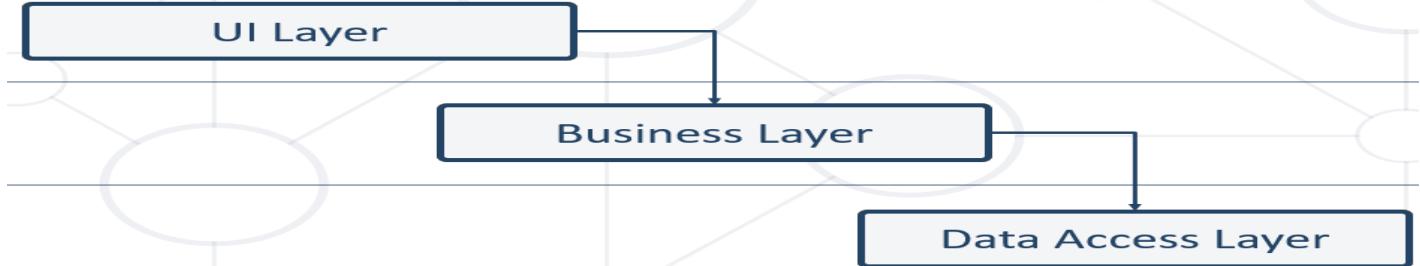
- Pros - No change in rest of the class; Very flexible

- **Cons** - Many parameters; Breaks the method signature

```
public class Copy {
    public void copyAll(Reader reader, Writer writer) {}
}
```

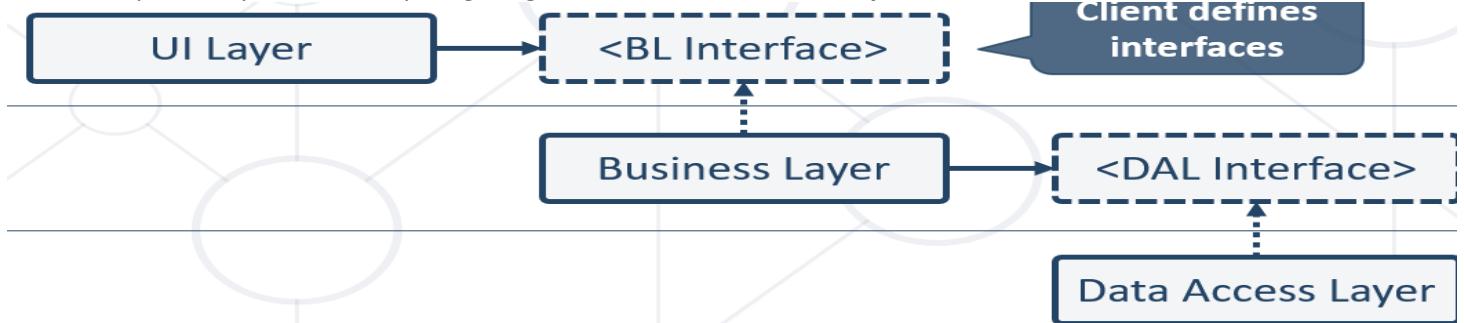
Layering (1)

- Traditional programming - **High-level** modules use **low-level** modules



Layering (2)

- Dependency Inversion Layering - High and **low-level** modules **depend on abstractions**



28.6. Exercises

Check the task with 1. Logger

Example in JAVA for working with Socket - използвайки програмата SocketTest - Test My Socket и след премахване на Firewall защитите на Windows

```
public class SocketAppender extends AppenderImpl {
    public SocketAppender(Layout layout) {
        super(layout);
    }

    @Override
    public void append(String time, String reportLevel, String message) {
        try {
            Socket socket = new Socket("localhost", 21);
            OutputStream outputStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(outputStream, true); //auto-flush is true
            writer.write(this.getLayout().format(time, reportLevel, message));
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

29. Reflection and Annotations

29.1. Reflection

Metaprogramming - Programming technique in which computer programs have the ability to treat programs as their data.

- Program can be designed to:
 - Read
 - Generate
 - Analyze
 - Transform
- **Modify itself while running**

Metadata – данни за source кода ни, информация за информацията

What is Reflection?

- The ability of a programming language to be its **own metalinguage** - с други думи, с JAVA език ще мога да модифицирам JAVA програми
- Programs can examine information about **themselves**

When to Use Reflection?

- Whenever we want:
 - Code to become more **extendible**
 - To **reduce code length** significantly
 - Easier **maintenance**
 - Easier **testing**

When Not to Use Reflection?

- If it is **possible** to perform an operation **without** using **reflection**, then it is preferable to **avoid using it**
- Cons from using Reflection
 - **Performance overhead**
 - **Security restrictions** – for example private is no longer private
 - Exposure of **internal logic**

29.2. Reflection API

Obtain its **java.lang.Class** object

The Class Object

- If you **know the name**

```
Class myObjectClass = MyObject.class;
```

- If you **don't know the name at compile time**

```
Class myClass = Class.forName(className); //You need fully qualified class name as String  
Class<?> aClass = Class.forName("com.mysql.cj.jdbc.Driver");
```

Class<Boolean> aClass = Boolean.class; - взима класа и прави променлива, с която може да работим след това.

Class Name

- Obtain **Class name**

- Fully qualified class name

```
String className = aClass.getName();
      ▪ Class name without the package name
String simpleClassName = aClass.get SimpleName();
```

Base Class and Interfaces

- Obtain **parent class**

```
Class className = aClass.getSuperclass();
      ▪ Obtain interfaces
Class[] interfaces = aClass.getInterfaces();
      ▪ Interfaces are also represented by Class objects in Java Reflection
      ▪ Only the interfaces specifically declared implemented by a given class are returned
```

```
public static void main(String[] args) {
    Class<Reflection> aClass = Reflection.class;

    System.out.println(aClass);

    System.out.println(aClass.getSuperclass());

    Class[] interfaces = aClass.getInterfaces();
    for (Class anInterface : interfaces)
        System.out.println(anInterface);

    //Reflection ref = aClass.newInstance(); //Deprecated since Java 9
    Reflection ref = aClass.getDeclaredConstructor().newInstance();
    System.out.println(ref);
}
```

29.3. Constructors, Fields and Methods

Constructors (1)

- Obtain **only public constructors**

```
Constructor[] ctors = aClass.getConstructors();
      ▪ Obtain all constructors - без значение какъв е access modifier-а им
Constructor[] ctors = aClass.getDeclaredConstructors();
      ▪ Get constructor by parameters
Constructor ctor = aClass.getConstructor(String.class);
```

Constructors (2)

- Get **parameter types**

```
Class[] parameterTypes = ctor.getParameterTypes(); // аргумент от тип String, int, char, etc.
      ▪ Instantiating objects 1 - using constructor
Constructor constructor = MyObject.class.getConstructor(String.class);
MyObject myObject = (MyObject)constructor.newInstance("arg1");
Object o = reClass.getDeclaredConstructor().newInstance();
```

- **Instantiating objects 2** - using constructor

Въпросчето показва, че все едно е конструктор от тип Object, башин, Wildcard

```
Constructor<?>[] constructors = blackBoxIntClass.getDeclaredConstructors();
try {
    constructors[0].setAccessible(true); // Change the behavior of the AccessibleConstructor
    constructors[0].newInstance();
```

```

} catch (InstantiationException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.printStackTrace();
}
}

```

Invoke the gotten constructor

```

BlackBoxInt blackBoxInt = constructor.newInstance();
String command = input.split(" ")[0];
int value = Integer.parseInt(input.split(" ")[1]);
currentMethod.setAccessible(true);
currentMethod.invoke(blackBoxInt, value); - изпълняваме върху обекта blackBoxInt със стойност
value

```

class blackBoxInteger.Main cannot access a member of class blackBoxInteger.BlackBoxInt with modifiers "private"

- **Instantiating objects 3** - using constructor – когато за да не забравим, искаме програмата сама да си създава обекта (но класа от който ще вдигаме инстанция за обекта трябва да сме го направили де 😊)

```

@Override
public Unit createUnit(String unitType) {
    Unit unit = null;
    try {
        Class<?> clazz = Class.forName(UNITS_PACKAGE_NAME + unitType); //където пакета е
        "barracksWars.models.units." //извиква класа на съответния unit(Pikeman/Horseman/Gunner, etc.)

        Constructor<?> constructor = clazz.getDeclaredConstructor(); //празен конструктор
        Constructor constructor =
            clazz.getDeclaredConstructor(String[].class, Repository.class, UnitFactory.class);

        constructor.setAccessible(true); //важи само за текущата try/catch конструкция
        constructor.newInstance(data, this.repository, this.unitFactory); //обект създаден с
        конструктор от 3 параметъра
        Object instance = constructor.newInstance(); //обект създаден с празен конструктор

        unit = (Unit) instance;
    } catch (ClassNotFoundException
             | NoSuchMethodException
             | InstantiationException
             | IllegalAccessException
             | InvocationTargetException e) {
        e.printStackTrace();
    }

    return unit;
}
switch (unitType) {
    case "Swordsman": unit = new Swordsman(); break;
    case "Archer": unit = new Archer(); break;
    case "Pikeman": unit = new Pikeman(); break;
    case "Horseman": unit = new Horseman(); break;
    case "Gunner": unit = new Gunner(); break;
}
}

```

Fields Name and Type

- Obtain **public** fields

```
Field field = aClass.getField("somefield");
Field[] fields = aClass.getFields();
```

- Obtain **all** fields – без значение какъв е access modifier

```
Field[] fields = aClass.getDeclaredFields();
```

- Get field **name and type**

```
String fieldName = field.getName(); // връща името на полето
Object fieldType = field.getType(); //връща типа данни на полето от тип wildcard башин Class<?>
String type = declaredField.getType().getSimpleClassName(); // връща типа данни на полето като стринг
```

Fields Set and Get

- Setting value for field

```
Class aClass = MyObject.class;
Field field = aClass.getDeclaredField("someField"); - връща и private поле ако е
MyObject objectInstance = new MyObject();
field.setAccessible(true); // Change the behavior of the AccessibleObject
Object value = field.get(objectInstance на дадения клас); //Get the field
field.set(objectInstance, value); //Set the field
The objectInstance parameter passed to the get and set method should be an instance of the class that owns the field.
```

Methods

- Obtain **public** methods

```
Method[] methods = aClass.getMethods();
Method method = aClass.getMethod("doSomething", String.class); // метод по име и параметър от тип
стринг
```

- Get methods without **parameters**

```
Method method = aClass.getMethod("doSomething", null); // метод по име и без параметри
```

Get Method, get method parameters and get method return type

- Obtain method **parameters** and **return type**

```
Class[] paramTypes = method.getParameterTypes();
Class returnType = method.getReturnType();
```

- Get methods with **parameters**

```
Method method = MyObject.class.getDeclaredMethod("doSomething", String.class);
Method method = MyObject.class.getMethod(nameMethod, void.class));
```

или

```
Method method = Arrays.stream(methods)
    .filter(m -> m.getName().equals("Ivane, ti si"))
    .findFirst().orElse(null);
```

Invoke the gotten method

```
Object returnValue = method.invoke(null, "arg1"); //null for static methods
```

```
BlackBoxInt blackBoxInt = (BlackBoxInt) constructor.newInstance(); //е инстанция на клас!!!
int param = 253;
try {
    method.setAccessible(true);
    method.invoke(blackBoxInt, param);
} catch (IllegalAccessException | InvocationTargetException e) {
```

```
    e.printStackTrace();
}
```

See the result of the Class field based on the operation done by the invoked method

```
if (innerValue != null) {
    try {
        System.out.println(innerValue.get(blackBoxInt)); // inner Value е поле на класа BlackBoxInt
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
```

29.4. Access Modifiers

- Obtain the **class modifiers** like this

```
int modifiers = aClass.getModifiers();
getModifiers() can be called on constructors, fields, methods
```

- Each modifier is a **flag bit** that is either set or cleared – побитови маски
- You can check the modifiers

```
int modifiers = aClass.getModifiers();
Modifier.toString(modifiers); - връща тип стринг текста – private/public/protected...
Modifier.isPrivate(modifiers);
Modifier.isProtected(modifiers);
Modifier.isPublic(modifiers);
Modifier.isStatic(modifiers);
```

Отпечатва modifier-те на всеки един метод

```
Class<Reflection> clazz = Reflection.class;
Method[] declaredMethods = clazz.getDeclaredMethods();
for (Method method : declaredMethods) {
    System.out.println(Modifier.toString(method.getModifiers()));
}
```

29.5. Проверка дали един метод е setter или getter

```
Class reflectionClass = ClassToExamine.class;
Method[] allMethods = reflectionClass.getDeclaredMethods();
for (Method method : allMethods) {
    int methodModifierNumber = method.getModifiers();
    if (isSetter(method) && !Modifier.isPrivate(methodModifierNumber)) {
        setters.add(method);
    } else if (isGetter(method) && !Modifier.isPublic(methodModifierNumber)) {
        getters.add(method);
    }
}
private static boolean isGetter(Method method) {
    if (!method.getName().startsWith("get")) {
        return false;
    }
    if (method.getReturnType() == void.class) {
        return false;
    }
    if (method.getParameterCount() != 0) {
```

```

        return false;
    }

    return true;
}

private static boolean isSetter(Method method) {
    if (!method.getName().startsWith("set")) {
        return false;
    }

    if (method.getReturnType() != void.class) {
        return false;
    }

    if (method.getParameterCount() != 1) {
        return false;
    }

    return true;
}

```

29.6. Arrays and Java reflection

- Creating arrays via Java Reflection – има общо с Generics

```
int[] intArray = (int[]) Array.newInstance(int.class, 3); // 3 е брой елементи
```

- Obtain parameter annotations

```
Array.set(intArray, 0, 123);
Array.set(intArray, 1, 456);
```

- Obtain fields and methods annotations

```
Class stringArrayComponentType = stringArrayClass.getComponentType();
```

29.7. Annotations

Анотациите носят допълнителна информация за полета, методи или цял клас. Докато самото поле, метод, конструктор или цял клас не може да я носи тази информация!

Анотацията дава **допълнителна** информация/значение/**поведение** на част от моя код.

- **Data holding class**
- **Describes parts of your code**
- Applied to: **Classes, Fields, Methods**, etc.

Annotation Usage

- To generate compiler messages or errors

@SuppressWarnings("unchecked") – не е сигурно, че ще мине

@Deprecated – остатяло/излиза от употреба/дава грешки/не е сигурно, че ще мине

- As tools
 - **Code generation tools**
 - **Documentation generation tools**
 - **Testing Frameworks**
- At runtime – **ORM, Serialization** etc.

Built-In Annotations (1)

- `@Override` – generates **compile time error** if the method does not override a method in a parent class
- ```
@Override
public String toString() {
 return "new toString() method";
}
```

### Built-In Annotations (2)

- `@SupressWarning` – turns off **compiler warnings** – да съпресваме на най-долно ниво, на самия ред само
- ```
@SupressWarnings(value = "unchecked") // annotation with value
public <T> void warning(int size) {
    T[] unchecked = (T[]) new Object[size]; // generates compiler warning
}
```

Built-In Annotations (3)

- `@Deprecated` – generates a **compiler warning** if the element is used (за стари излизащи от употреба неща)

Creating Annotations

- `@interface` – the keyword for annotations

```
public @interface MyAnnotation {
    String myValue() default "default"; // Annotation element
}

@MyAnnotation(myValue = "value") // Skip name if you have only one value named "value"
public void annotatedMethod() {
    System.out.println("I am annotated");
}
```

Annotation Elements

- Allowed types for annotation elements:
 - Primitive types (`int`, `long`, `boolean`, etc.)
 - `String`
 - `Class`
 - `Enum`
 - `Annotation`
 - `Arrays` of any of the above

Meta Annotations – `@Target`

- **Meta annotations** annotate annotations
- `@Target` – specifies where the annotation is applicable

```
@Target(ElementType.FIELD) // Used to annotate fields of the class only
public @interface FieldAnnotation {
}
```

- Available element types – **CONSTRUCTOR**, **FIELD**, **LOCAL_VARIABLE**, **METHOD**, **PACKAGE**, **PARAMETER**, **TYPE**

```
@Target(ElementType.TYPE) // Used to annotate the whole class
@Target(ElementType.METHOD) // Used to annotate the method of the class only
```

Meta Annotations – @Retention – задържане/ангажиране

- **@Retention** – specifies where annotation is available – докога да се пази

```
@Retention(RetentionPolicy.RUNTIME) //you can get info at runtime and the created annotation will  
not be deleted after we enter from compiling to running stage
```

```
public @interface RuntimeAnnotation {  
    // ...  
}
```

- Available retention policies – **SOURCE, CLASS, RUNTIME**

- Create Annotation

New java class ->

```
public @interface FieldAnnotation {  
}
```

```
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Subject{  
    String[] categories();  
}
```

- Obtain class annotations

```
Annotation[] annotations = aClass.getAnnotations();  
Annotation annotation = aClass.getAnnotation(MyAnno.class);  
Annotation[] annotations = reflectionClass.getDeclaredAnnotations();
```

- Obtain parameter annotations – using matrix – двойна анотация

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

- Obtain fields and methods annotations

```
Annotation[] fieldAnots = field.getDeclaredAnnotations();  
Annotation[] methodAnot = method.getDeclaredAnnotations();
```

Accessing Annotation (1)

- Some annotations can be accessed at runtime

```
@Retention(RetentionPolicy.RUNTIME)  
public @interface Author {  
    String name(); //тук е метод  
    String name() default "Unknown"; //можем да зададем и default-на стойност  
}
```

```
@Author(name = "Gosho") // но тук не е метод  
public class AuthoredClass {  
    public static void main(String[] args) {  
        Class cl = AuthoredClass.class;  
        Author author = (Author) cl.getAnnotation(Author.class);  
        System.out.println(author.name());  
    }  
}
```

Accessing Annotation (2)

- Some annotations can be accessed **at runtime**

```
@Retention(RetentionPolicy.RUNTIME)  
public @interface Author {  
    String name();  
}
```

```

@Author(name = "Gosho") // но тук не е метод
public class AuthoredClass {
    Class cl = AuthoredClass.class;
    Annotation[] annotations = cl.getAnnotations();
    for (Annotation annotation : annotations) {
        if (annotation.annotationType().equals(Author.class)) {
            Author author = (Author) annotation;
            System.out.println(author.name());
        }
    }
}

```

Други примери

```

@Retention(RetentionPolicy.RUNTIME)
public @interface Entity {
    String name();
}

private List<String> getColumnsWithoutId(Class<?> aClass) {
    List<String> collect = Arrays.stream(aClass.getDeclaredFields())
        .filter(f -> !f.isAnnotationPresent(Id.class))
        .filter(f -> f.isAnnotationPresent(Column.class)) //само ги филтрирај - тези полета,
които са анотирани с Column анатомия
        .map(f -> f.getAnnotationsByType(Column.class)) //след като са налични полетата, ги
вземи
        .map(a -> a[0].name()) //Вземи името на полето - само веднъж имаме върху класа User
анотация с Entity анатомия, или анатомия Entity се използва само на един клас за момента
        .collect(Collectors.toList());

    return collect;
}

```

Първо анотиране в случая

```

annotationsByType = {Column[1]@2574}
  0 = {$Proxy3@2580} "@annotations.Column(name=id)"
    h = {AnnotationInvocationHandler@2584}
      type = {Class@2568} "interface annotations.Column"... NavigableSet
      memberValues = {LinkedHashMap@2585} size = 1
        "name" -> "id"
      memberMethods = null

```

```

@Entity(name = "users")
public class User {
    @Id
    @Column(name = "id")
    private long id;

    @Column(name = "username")
    private String username;

    @Column(name = "age")
    private int age;

    @Column(name = "registration_date")

```

```

private LocalDate registrationDate;

//Add new field when adding column to the database - we change only in the User class
@Column(name = "last_logged_in")
private LocalDate lastLoggedIn;
}

```

Още един пример:

```

@Column(name = "registration_date")
private LocalDate registrationDate;

```

```

//from SQL type we convert to JAVA data type - for each field - обратното на getSQLType метода
private void fillField(Field declaredField, ResultSet resultSet, E resultEntity) throws
SQLException,
    IllegalAccessException {
    Class<?> fieldType = declaredField.getType();
    //String fieldName = declaredField.getName(); //връща името на полета на изкуствената
инстанция, която е взела имена на полетата от SQL базата/таблицата или "registration_date"
    String fieldName = declaredField.getAnnotationsByType(Column.class)[0].name(); //чрез
използване на анотация връща името на полето от изкуствената инстанция на класа или
registrationDate

```

29.8. Dependency Injection Container – example!!!

Където няма нужда да се подават повече параметри на конструктора, ги избягваме

```

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Inject {
}

public abstract class Command implements Executable {
    private String[] data;

    protected Command(String[] data){
        this.data = data;
    }
}

//addUnit command
public class Add extends Command {
    @Inject private Repository repository;
    @Inject private UnitFactory factory;

    public Add(String[] data) { super(data);}
    @Override
    public String execute() {
        String unitType = this.getData()[1];
        Unit unitToAdd = this.factory.createUnit(unitType);
        this.repository.addUnit(unitToAdd);
        return unitType + " added!";
}

```

```
}
```

```
//retire command
public class Retire extends Command {
    @Inject private Repository repository;
    public Retire(String[] data) {super(data);}
@Override
    public String execute() { DO SOMETHING }
}

//report command
public class Report extends Command {
    @Inject private Repository repository;
    public Retire(String[] data) {super(data);}
@Override
    public String execute() { DO SOMETHING }
}

//fight command
public class Fight extends Command{
    public Fight(String[] data) { super(data);}
@Override
    public String execute() { DO SOMETHING }
}

public interface Executable {
    String execute();
}

public class CommandInterpreterImpl implements CommandInterpreter {
    private final static String PACKAGE_NAME = "barracksWars.core.commands.";
    private Repository repository;
    private UnitFactory unitFactory;

    public CommandInterpreterImpl(Repository repository, UnitFactory unitFactory) {
        this.repository = repository;
        this.unitFactory = unitFactory;
    }

    @Override
    public Executable interpretCommand(String[] data, String commandName) { //метод от интерфейса
CommandInterpreter
        Executable executable = null;

        String command = getCorrectClassName(data[0]); //първата буква я прави главна - метод
        try {
            Class clazz = Class.forName(PACKAGE_NAME + command);
            Constructor constructor = clazz.getDeclaredConstructor(String[].class); //конструктор,
който да поема масив от стрингове
            constructor.setAccessible(true);
            executable = (Executable) constructor.newInstance(new Object[]{data}); //може и само
data да му дадем, но има конфликт дали е varargs или е масив
            populateDependencies(executable); //метода, който работи с анотациите
        } catch (ClassNotFoundException | InstantiationException
                | IllegalAccessException | InvocationTargetException | NoSuchMethodException e) {
        }

        return executable;
    }
}
```

```

private void populateDependencies(Executable executable) {
    Field[] exeFields = executable.getClass().getDeclaredFields();
    Field[] currentClazFields = this.getClass().getDeclaredFields(); //връща или
this.repository или this.unitFactory - трябват ни 0, 1 или 2 анотациоанни съвпадения
    for (Field requiredField : exeFields) {
        Inject annotation = null;
        try {
            annotation = requiredField.getAnnotation(Inject.class); //имаме ли съвпадение с
анотацията
        } catch (ClassCastException e){
            continue;
        }

        //if requiredField must be injected
        for (Field currentClazField : currentClazFields) {
            if (currentClazField.getType().equals(requiredField.getType())) {
                requiredField.setAccessible(true);
                try {
                    requiredField.set(executable, currentClazField.get(this)); //на инстанцията
на текущия клас, задаваме такива активни полета щото да има съвпадение с анотациите
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

public class Engine implements Runnable {
    private CommandInterpreter commandInterpreter;

    public Engine(CommandInterpreter commandInterpreter) {
        this.commandInterpreter = commandInterpreter;
    }

    @Override
    public void run() {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        while (true) {
            try {
                String input = reader.readLine();
                String[] data = input.split("\\s+");
                String commandName = data[0];
                Executable currExec = this.commandInterpreter.interpretCommand(data, commandName);
                //връща такъв обект, който има съответните полета спрямо съвпадение на съответните анотации от
                класовете или Add или Report или Fight или Retire, за да може да се изпълни метода execute()
                String result = currExec.execute(); //връща изпълнение или на добавяне на елемент, или на
                пенсиониране, или на report, или на fight
                if (result.equals("fight")) {
                    break;
                }
                System.out.println(result);
            } catch (RuntimeException e) {
                System.out.println(e.getMessage());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

    }
}

public class Main {

    public static void main(String[] args) {
        Repository repository = new UnitRepository();
        UnitFactory unitFactory = new UnitFactoryImpl();
        CommandInterpreter commandInterpreter =
new CommandInterpreterImpl(repository, unitFactory);

        Runnable engine = new Engine(commandInterpreter);
        engine.run();
    }
}

```

30. Exception Handling

30.1. What Are Exceptions?

Handling Errors During the Program Execution

The Throwable Class

- Exceptions in Java are **objects**
- The **Throwable class** is a base for all exceptions in JVM
- Contains information for the cause of the error
 - **Message** – a text description of the exception
 - **StackTrace** – the snapshot of the stack at the moment of exception throwing

Types of Exceptions

- Java exceptions inherit from **Throwable**
- Below **Throwable** are:
 - **Error** - **not expected** to be caught under normal circumstances from the program
Example - **StackOverflowError**
 - **Exception**
 - Used for exceptional conditions that user programs should catch
 - User-defined exceptions

Exceptions are two types:

Checked - an exception that is checked (notified) by the compiler at compilation-time - also called as **Compile Time exceptions**

```

public static void main(String args[]) {
    File file = new File("E:/file.txt");
    FileReader fr = new FileReader(file); // FileNotFoundException
}

```

Checked exceptions are *subject to the Catch or Specify Requirement*. All exceptions are checked exceptions, except for those indicated by **Error**, **RuntimeException**, and their subclasses.

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

Unchecked - Runtime Exceptions – an exception that occurs at the time of execution

These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the

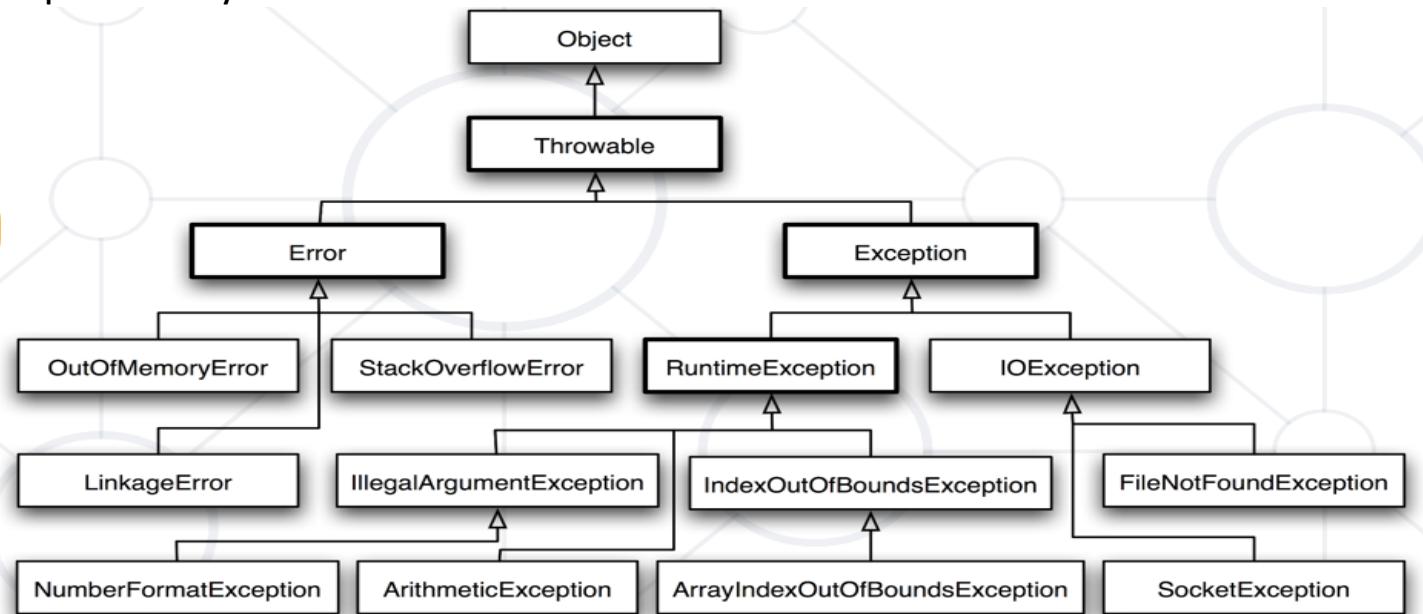
application described previously that passes a file name to the constructor for `FileReader`. If a logic error causes a null to be passed to the constructor, the constructor will throw `NullPointerException`. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Unchecked - Error— an exception that occurs at the time of execution

The second kind of exception is the `error`. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw `java.io.IOException`. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by `Error` and its subclasses.

Exception Hierarchy



Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions. In general, this is not recommended. The section [Unchecked Exceptions — The Controversy](#) talks about when it is appropriate to use unchecked exceptions.

Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

30.2. Handling Exceptions

- In Java exceptions can be handled by the `try-catch` construction

```
try {  
    // Do some work that can raise an exception  
} catch (SomeException) {  
    // Handle the caught exception  
}
```

- `catch` blocks can be used multiple times to process different exception types

- When catching an exception of a particular class, **all its inheritors (child exceptions) are caught too**, e.g. – т.e. хваща exception-а `IndexOutOfBoundsException` и всички негови деца/наследници

```
try {
    // Do some work that can cause an exception
} catch (IndexOutOfBoundsException ae) {
    // Handle the caught arithmetic exception
}

▪ Handles IndexOutOfBoundsException and its descendants ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException

try {
    Integer.parseInt(str);
} catch (Exception ex) { //should be last
    System.out.println("Cannot parse the number!");
} catch (NumberFormatException ex) { //should be first
    System.out.println("Invalid integer number!");
}
```

Handling All Exceptions – не е добра идея да ползваме за всички

- For handling all exceptions (even unmanaged) use the construction:

```
try {
    // Do some work that can raise any exception
} catch (Exception ex) {
    // Handle the caught exception
}
```

The Try-finally Statement

```
try {
    // Do some work that can cause an exception
    return;
} catch (Exception ex) {
    // Handle the caught exception
}
finally {
    // This block will always execute
}
```

30.3. Throwing Exceptions

- Exceptions are thrown (raised) by the `throw` keyword
- Used to notify the calling code in case of an error or unusual situation
- When an exception is thrown:
 - The program execution stops
 - **The exception travels over the stack**
 - Until a matching `catch` block is reached to handle it
- Unhandled exceptions display an error message

Using Throw Keyword

- Throwing an exception with an error message:

```
throw new IllegalArgumentException("Invalid amount!");
```

- Exceptions can accept `message` and `cause` – използва се когато искаме да преобразуваме вида Exception:

```

try {
...
} catch (SQLException sqlEx) {
    throw new IllegalStateException("Cannot save invoice.", sqlEx);
}


- Note: if the original exception is not passed, the initial cause of the exception is lost

```

Re-Throwing Exceptions

- Caught exceptions can be re-thrown again:

```

try {
    Integer.parseInt(str);
} catch (NumberFormatException ex) {
    System.out.println("Parse failed!"); //или запиши в базата данни, и след това следващия по
    //веригата да го хване наново
    throw ex; // Re-throw the caught exception
}

```

Използвайки конзолния поток за грешки `System.err` вместо конзолния поток `System.in` или `System.out`

```

catch (IllegalArgumentException ex) {
    System.err.println("Error: " + ex.getMessage());
    ex.printStackTrace();
}

```

30.4. Best Practices

Хитринка – пишем грешката да е по-обща. IntelliJ връща конкретния тип грешка заради полиморфизът. И вече може да си декларираме верният вид Exception в кода!!!

Using Catch Block

- **Catch** blocks should:
 - Begin with the exceptions lowest in the hierarchy
 - Continue with the more general exceptions
 - Otherwise a compilation error will occur
- Each **catch** block should handle only these exceptions which it expects
 - If a method is not competent to handle an exception, it should leave it unhandled
 - Handling all exceptions disregarding their type is a popular **bad practice** (anti-pattern)!

Choosing the Exception Type

- When an application attempts to use **null** in a case where an object is required – **NullPointerException**
- An array has been accessed with an illegal index – **ArrayIndexOutOfBoundsException**
- An index is either negative or greater than the size of the string – **StringIndexOutOfBoundsException**
- Attempts to convert an inappropriate string to one of the numeric types – **NumberFormatException**
- When an exceptional arithmetic condition has occurred – **ArithmaticException**
- Attempts to cast an object to a subclass of which it is not an instance – **ClassCastException**
- A method has been passed an illegal or inappropriate argument - **IllegalArgumentException**

Best Practices examples

- When raising an exception, always pass to the constructor a **good explanation message**
- When throwing an exception always pass a good description of the problem
 - The exception message should explain what causes the problem and how to solve it
 - Good: "Size should be integer in range [1...15]"

- Good: "Invalid state. First call Initialize()"
 - Bad: "Unexpected error"
 - Bad: "Invalid argument"
- Exceptions can decrease the application performance
 - Throw exceptions only in situations which are really exceptional and should be handled
 - Do not throw exceptions in the normal program control flow
 - JVM could throw exceptions at any time with no way to predict them
 - E.g. `StackOverflowError`

Да избягваме когато можем използването на Exceptions

Ако можем да зададем на кода примерно, че може да не го инициализираме, то го правим

```
Optional<Integer> age; //може да го инициализираме, а може и да не го инициализираме
age.isEmpty();
age.isPresent();
```

В някои езици може да се използва следната структура:

`Either<Left, Right>` - ако е инициализирано върни `Right`, в противен случай върни `Left`
`Either<String, Integer>` - ако няма грешка върни `Integer`, в противен случай върни грешката `String`

30.5. Custom Exceptions

- Custom exceptions inherit an exception class (commonly – `Exception`)

```
public class TankException extends Exception {
    public TankException(String msg) {
        super(msg);
    }

    public TankException(String message, Exception/Throwable cause) {
        super(message, cause);
    }

    @Override
    public void printStackTrace(PrintStream s) {
        super.printStackTrace(s);
    }

    @Override
    public String getMessage() {
        return super.getMessage();
    }

    @Override
    public synchronized Throwable getCause() {
        return super.getCause();
    }
}
```

- Thrown just like any other exception

```
throw new TankException("Not enough fuel to travel");
```

- Може да използваме директно и по този начин

```
public static void main(String[] args) {
    try {
        throw new TankException("Not enough fuel to travel");
    } catch (TankException e) {
```

```

        System.out.println(e.getMessage());
    }
}

• RuntimeException
public class ValidationException extends RuntimeException {

    public ValidationException(String reason) {
        super(reason);
    }
}

```

31. Debugging

31.1. What is Debugging?

- The process of locating and fixing or bypassing **bugs** (errors) in computer program code
- To **debug** a program:

Start with a **problem**

Isolate the **source** of the problem

Fix it

- **Debugging tools** (called **debuggers**) help identify **coding errors** at various development stages

▪ **Testing**

A means of initial detection of errors

- **Debugging**

A means of diagnosing and correcting the root causes of errors that have already been detected

▪ **Legacy code**

You should be able to debug code that is written years ago

Debugging Philosophy

- Debugging can be viewed as one big **decision tree**
 - Individual nodes represent **theories**
 - **Leaf nodes** (всеки клон/връх който няма дете/наследник) represent possible **root causes**
 - Traversal of tree boils down to process state **inspection**
 - Minimizing time to resolution is **key**
 - Careful traversal of the decision tree
 - Pattern recognition
 - Visualization and ease of use helps minimize time to resolution

Писане на консистентен код – един проблем по един и същи начин - pattern

31.2. IntelliJ IDEA Debugger

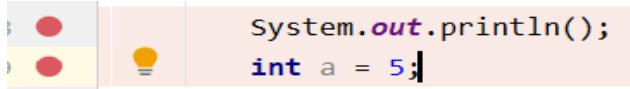
В режим на дебъгване, се зареждат куп допълнителни неща, които в нормалния режим на работа на кода, няма да се заредят.

Най-добре да използваме до 2-3 breakpoint-а за дебъгване, а не повече!!!

- IntelliJ IDE gives us a lot of **tools** to **debug** your application

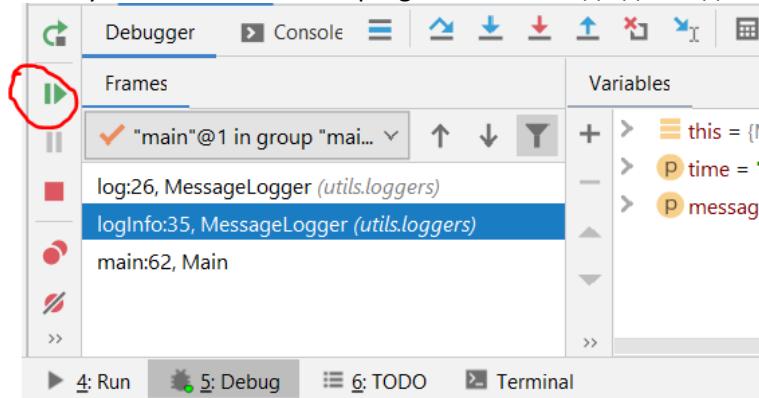
Adding breakpoints

Понякога не е добре да си слагаме breakpoint на нещо, което печатаме на конзолата, но да сложим breakpoint на променлива присвояваща число винаги е ок.



Visualize the program flow

Този бутон показва Resume program – или отиди до следващия (същи) breakpoint



Control the flow of execution

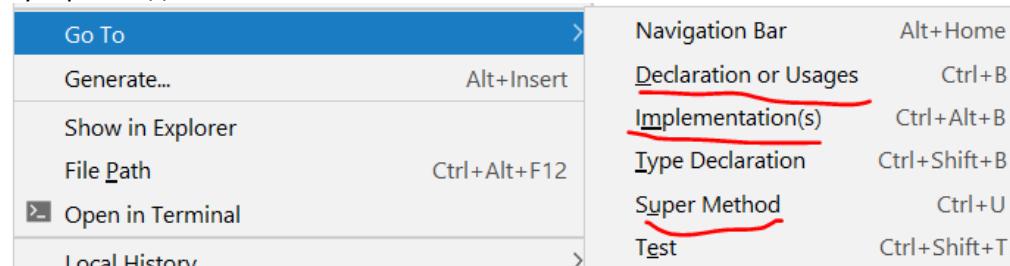
Да слагаме breakpoint на определените места

Data tips

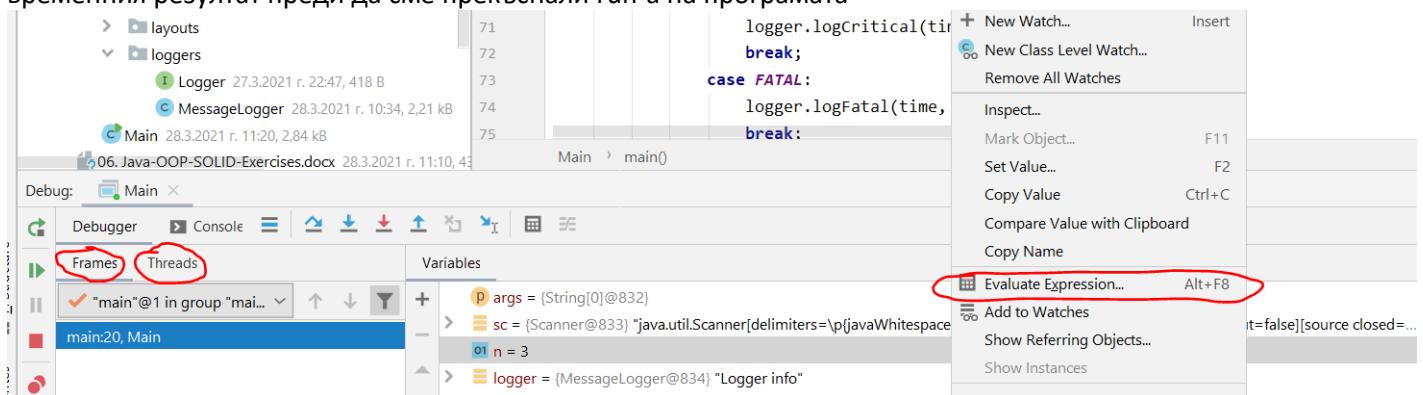
Декларации и употреби

Имплементации

Супер метод

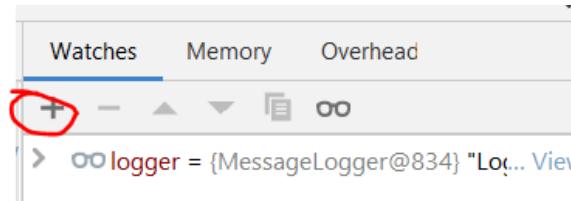
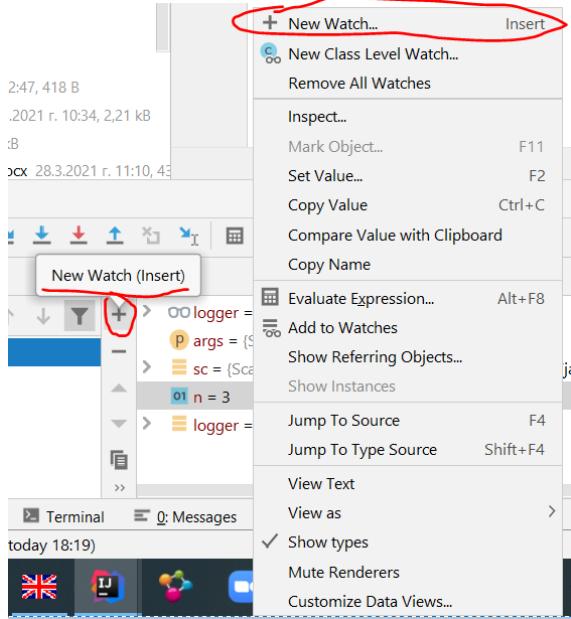


Evaluate expression – казва какво ще ретърне навън още преди да го е ретърнало – можем да го сравним с временния резултат преди да сме прекъснали run-а на програмата

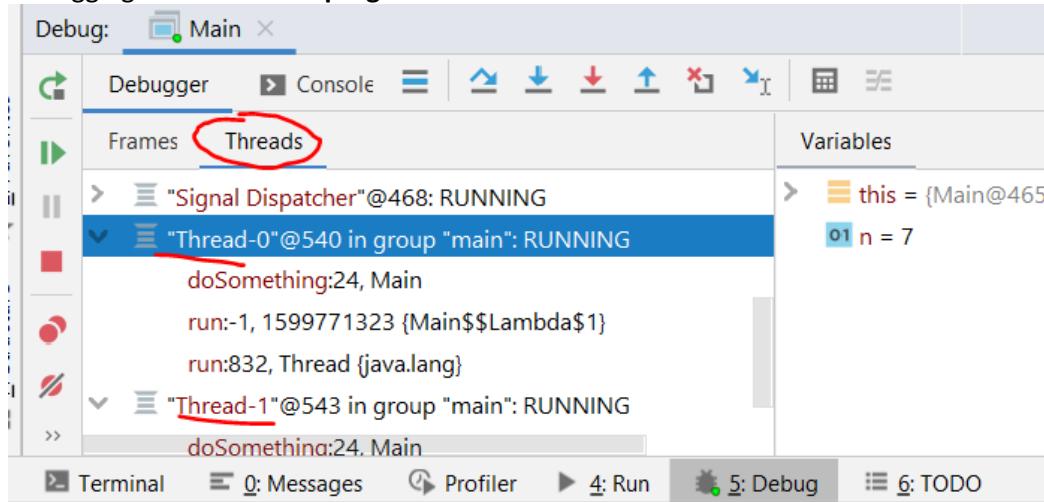


Watch variables

New watch variable може да добавяме – **ако добави в раздел Watches – винаги следим какво се случва с дадената променлива – watch-а не е сред многото други променливи за следене, а е на отделно място**



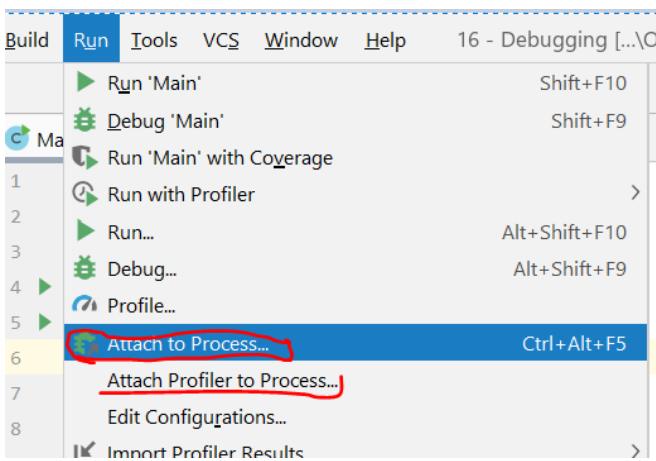
Debugging multithreaded programs



And many more...

How to Debug a Process

- Option 1 - Starting a process under the IntelliJ debugger
- Option 2 - Attaching to an already running process – закачаме се за вече пуснат процес, примерно от друга програма на операционната система
 - Without a solution loaded you can still debug
 - Useful when solution isn't readily available
 - **Ctrl + Alt + F5**



Debugging a Project

Има няколко начина за стартиране на дебъгера. Когато сме в непознат проект, този начин може да използваме:

- Right click in **main** method, Debug '{class}.main()' (Main.main примерно)

Shift + F9 is a shortcut

- Easier access to the source code and symbols since its loaded in the solution
- Certain differences exist in comparison to debugging an already running process

Debug Windows

- Debug Windows are the means to introspect on the state of a process
- Opens a new window with the selected information in it
- Window categories

Frames / Threads

Variables

Watches

- Accessible from Debug window

Debugging Toolbar

- Convenient shortcut to common debugging tasks

Step over – F8 - Steps over the current line of code and takes you to the next line even if the highlighted line has method calls in it. The implementation of the methods is skipped, and you move straight to the next line of the caller method.

Step into – F7 – ред по ред - Steps into the method to show what happens inside it. Use this option when you are not sure the method is returning a correct result.

Force Step Into – through the method calls - Alt + Shift + F7 – ако не ни взлиза дебъгера в даден метод

Step Out – Shift + F8 – излизаме от даден метод/стек, за да се прехвърлим на друг - Steps out of the current method and takes you to the caller method.



Continue – (resume program F9) | ➤

Break - ⚡

Breakpoints

The screenshot shows the IntelliJ IDEA interface during debugging. The code editor displays Java code with two red circles highlighting specific breakpoints. The 'Breakpoints' tool window on the left shows three breakpoints: 'RegistrationControl', 'ViewItemsController', and 'ComputerStoreApplication'. The 'Variables' tool window at the bottom shows the variable 'this' with the value '{ValidPriceValidator@12645}'. A red box highlights the 'Run to Cursor Alt+F9' button in the 'Variables' window. The status bar at the bottom indicates the application is 'RUNNING'.

- By default, an app will run uninterrupted (and stop on exception or breakpoint) when in debugging regime
- Debugging is all about looking at the **state of the process**
- Controlling execution allows:

Pausing execution

Resuming execution

- Options and settings is available via

File-> Settings/Preferences -> Build, Execution and Deployment (Ctrl + Alt + S):

Debugger -> Data Views -> **Java**

Compiler -> **Java Compiler**

- Settings for project structure via

File -> Project Structure (Ctrl + Shift + Alt + S)

31.3. Breakpoints

- Ability to stop execution based on certain criteria is key when debugging

When a function is hit

When data changes

When a specific thread hits a function

Much more...

Натискаме с десен бутон върху червената точка/breakpoint, като задаваме някакво условие да бъде изпълнено, все едно си пишем код нормално:

AppenderImpl 28.3.2021 г. 11:01, 1,46 KB
 ConsoleAppender 28.3.2021 г. 10:48, 513 B
 FileAppender 28.3.2021 г. 10:51, 758 B
 SocketAppender 28.3.2021 г. 11:25, 825 B

enums
 helpOutFile
 layouts
 loggers
 Logger
 Message

Main.java:58

Enabled
 Suspend: All Thread
 Condition:
 reportLevel.equals(ReportLevel.CRITICAL)

Done

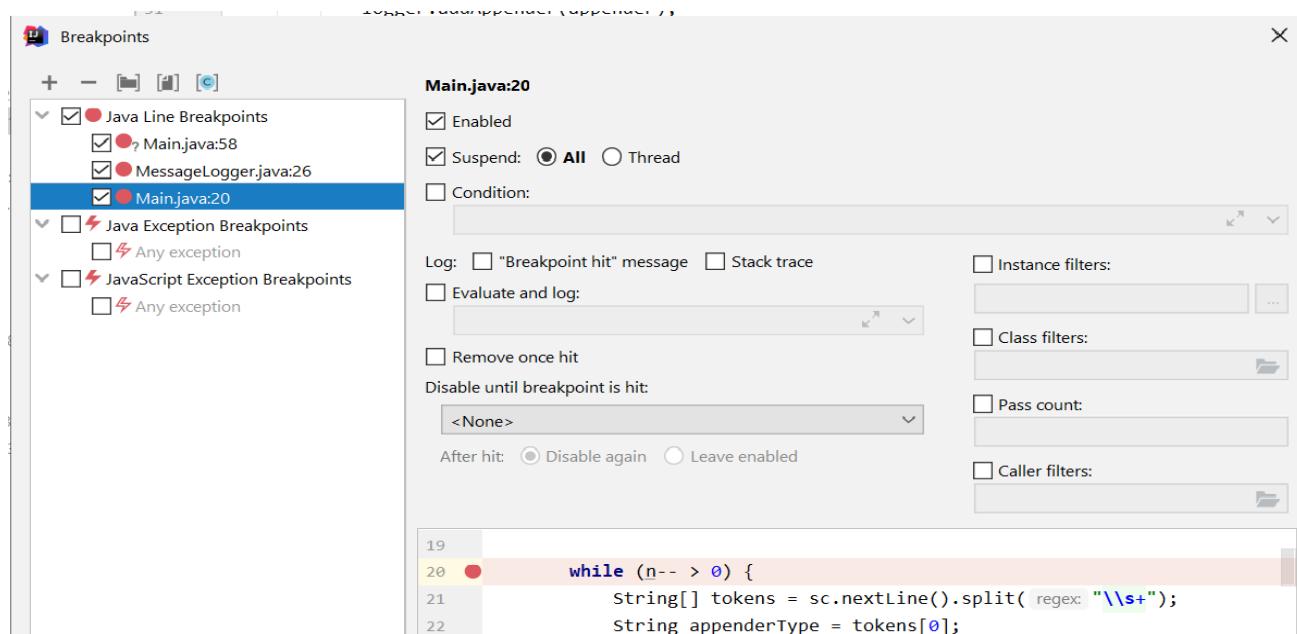
```

56
57
58 while (! END.equals(input)) {
59     String[] tokens = input.split( regex: "\\\\" );
60     ReportLevel reportLevel = ReportLevel.valueOf(tokens[0]);
61     String time = tokens[1];
62     String level = tokens[2];
63     if (level.equals("INFO")) {
64         logger.logInfo(time, message);
65     } else if (level.equals("WARNING")) {
66         logger.LogWarning(time, message);
67     } else if (level.equals("ERROR")) {
68         logger.logError(time, message);
69         break;
70     } case CRITICAL:
71         logger.logCritical(time, message);
72         break;
73     case FATAL:
74         logger.logFatal(time, message);
75         break;
    }
    
```

Менажиране на breakpoints -> More (Ctrl + Shift + F8)

Имаме бърз достъп до всички breakpoints в нашия проект

- Adding breakpoints
- Removing or **disabling** breakpoints



- Stops execution at a specific instruction (line of code)

Can be set using:

Ctrl + F8 shortcut - слага червената точка

Clicking on the left most side of the source code window - или просто натискаме с ляв бутон от лявата страна, за да сложим червената точка

- By default, the breakpoint will hit every time execution reaches the line of the code
- Additional capabilities: condition, hit count, value changed, when hit, filters

Breakpoints не работят:

- В някои случаи – затова използваме **Force Step Into**

- Във всички случаи на асинхронна среда

31.4. Stream debugging / Дебъгване на stream



Trace current stream chain

31.5. Data Inspection

Variables and Watches Windows – казахме ги по-горе

- Allows you to inspect various states of your application
- Several different kinds of "predefined" watches window
- "Custom" watches windows also possible

Contains only variables that you choose to add

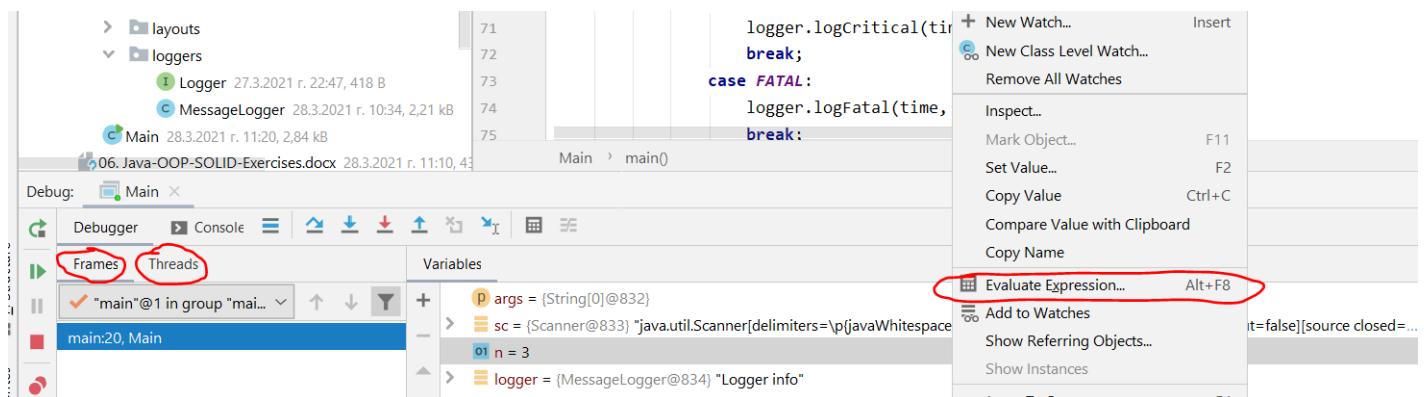
Right click on the variable and select "Add to Watches"

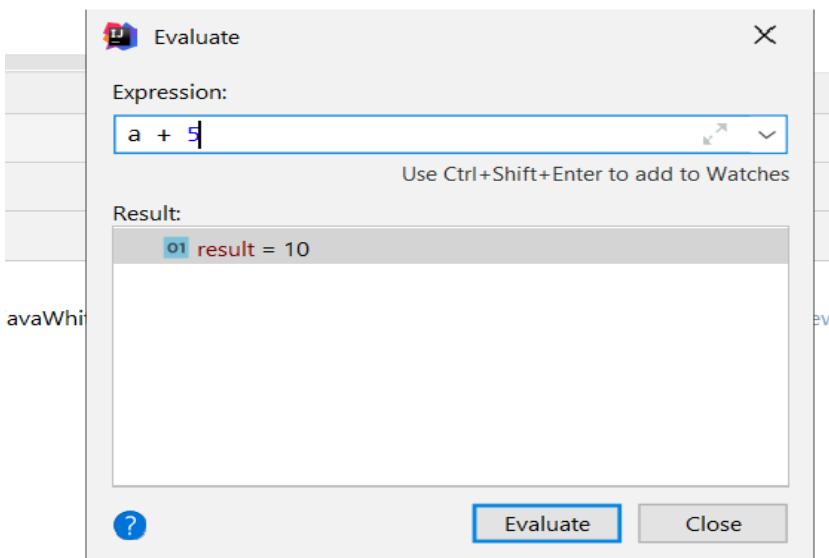
Write the variable name in Watches window

Evaluate Expression Window - казахме го по-горе

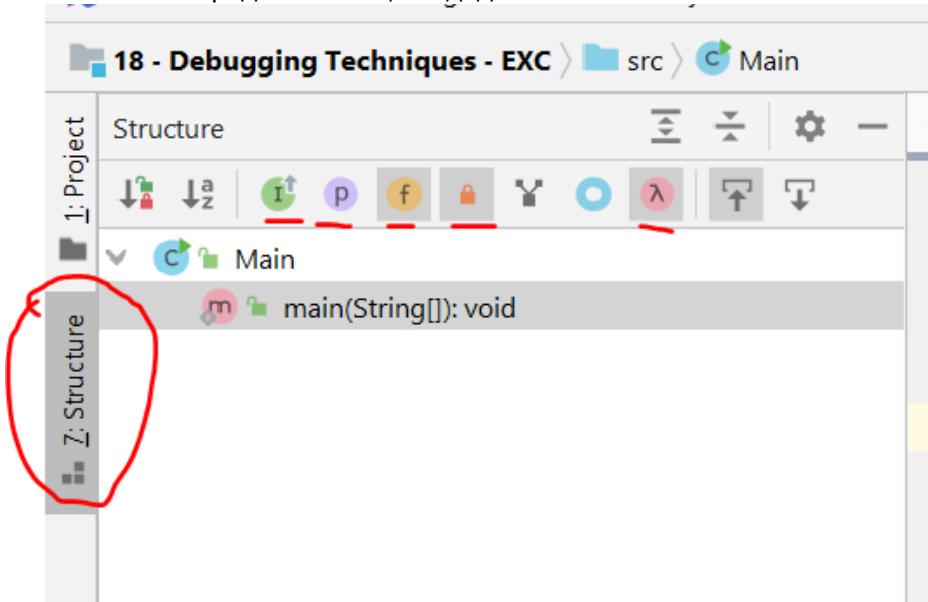
- Enables to evaluate expressions and code fragments in the context of a stack frame
- Also evaluate operator expressions, lambda expressions, and anonymous classes
- Shortcut – Alt + F8

Evaluate expression – казва какво ще ретърне навън още преди да го е ретърнало – можем да го сравним с временния резултат преди да сме прекъснали run-а на програмата – СМЯТА НИ НЕЩА КОИТО НИЕ ИСКАМЕ ДА РАЗБЕРЕМ (примерно някакви променливи от кода като ги съберем какъв резултат дават)





Показване на определени неща от даден клас:



31.6. Finding a Defect

- Stabilize the error
- Locate the source of the error
 - Gather the data
 - Analyze the data and form hypothesis
 - Determine how to prove or disprove the hypothesis
- Fix the defect
- Test the fix
- Look for similar errors

Tips

- Use all available data

- Refine the test cases
- Check unit tests
- Use available tools
- Reproduce the error in several different ways
- Generate more data to generate more hypotheses
- Use the results of negative tests
- Brainstorm for possible hypotheses
- Narrow the suspicious region of the code
- Be suspicious of classes and routines that have had defects before
- Check code that's changed recently
- Expand the suspicious region of the code
- Integrate incrementally
- Check for common defects
- Talk to someone else about the problem
- Take a break from the problem

Fixing a Defect

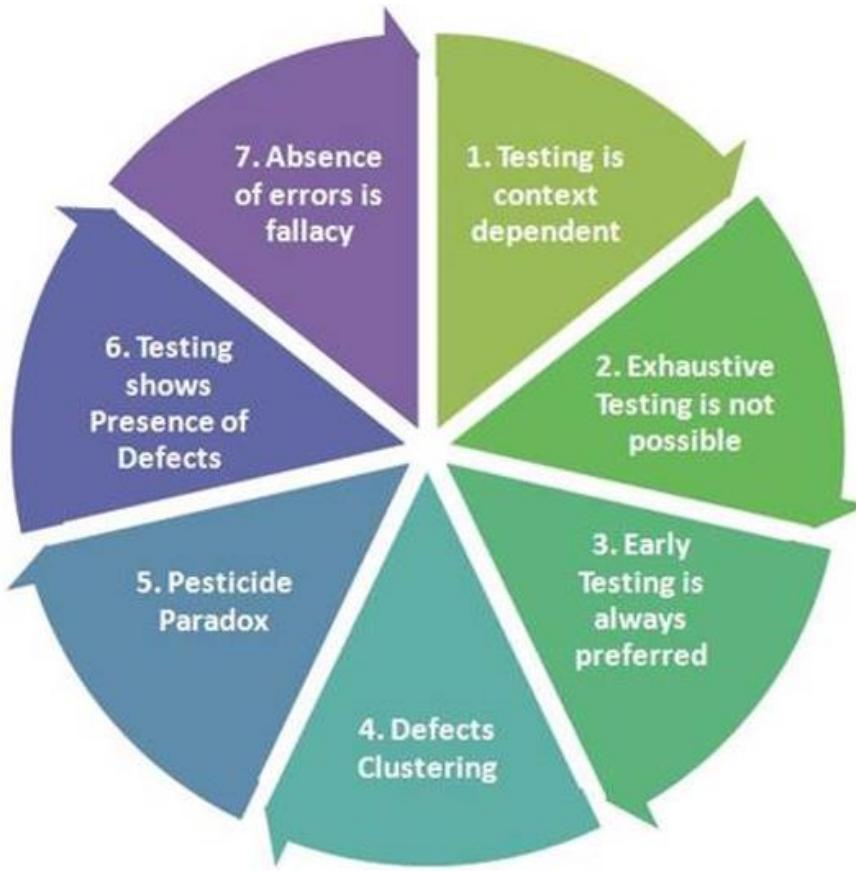
- Understand the problem before you fix it
- Understand the program, not just the problem
- Confirm the defect diagnosis
- Relax
- Save the original source code
- Fix the problem, not the symptom
- Make one change at a time
- Add a unit test that expose the defect
- Look for similar defects

Psychological Considerations

- Your ego tells you that your code is good and doesn't have a defect even when you've seen that it has
- How "psychological set" contributes to debugging blindness
 - People expect a new phenomenon to resemble similar phenomena they've seen before
 - Do not expect anything to work "by default"
 - Do not be too devoted to your code – establish psychological distance

32. Unit Testing

32.1. Seven Testing Principles



Testing is context dependent(1)

Testing is done differently in **different contexts**

- Example:

Safety-critical software is tested **differently** from an e-commerce site

Exhaustive testing is impossible (2)

All combinations of inputs and preconditions are usually almost **infinite number**

Testing everything is not feasible

Except for trivial cases

Risk analysis and priorities should be used to focus testing efforts

Early testing is always preferred (3)

Testing activities shall be started as early as possible

And shall be focused on defined objectives

The later a bug is found – the more it costs!

Defect clustering (4)

Testing effort shall be focused **proportionally**

To the expected and later observed defect density of modules

A **small number** of modules usually contains **most of the defects** discovered

Responsible for most of the operational failures

Pesticide paradox (5)

Same tests repeated **over and over again** tend to **lose their effectiveness**

Previously **undetected** defects remain **undiscovered**

New and modified test cases should be developed

Testing shows presence of defects (6)

Testing can **show that defects are present**

Cannot prove that there are no defects

Appropriate testing **reduces** the probability for defects

Absence-of-errors fallacy (7) - заблуда

Finding and **fixing** defects itself does not help in these cases:

The system built is unusable

Does not fulfill the users' needs and expectations

32.2. What is Unit Testing

Manual Testing – не предвижда всички случаи, и всеки път ръчно го въвеждаме теста

- Not **structured**
- Not **repeatable**
- Can't **cover** all of the code
- Not as **easy** as it should be

- We need a **structured approach** that:

Allows **refactoring**

Reduces the **cost of change**

Decreases the number of **defects** in the code

- Bonus:

Improves **design**

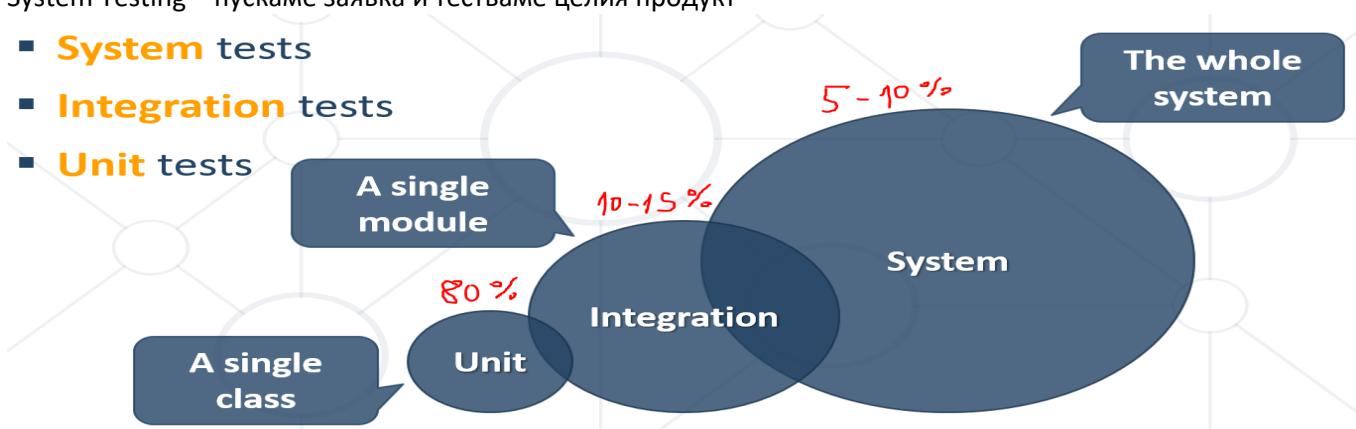
Automated Testing

Unit Testing – изолиран най-малък код / single responsibility

Integration Testing – няколко юнита заедно

System Testing – пускаме заявка и тестваме целия продукт

- **System tests**
- **Integration tests**
- **Unit tests**



32.3. JUnit framework в Java ще използваме

- Maven Repository (<https://mvnrepository.com/>) – всички неща, които има в Maven

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
```

```

<artifactId>maven-compiler-plugin</artifactId>
<configuration>
    <source>8</source>
    <target>8</target>
</configuration>
</plugin>
</plugins>
</build>

```

- Junit 4.12 – <https://mvnrepository.com/artifact/junit/junit/4.12> - в Maven е и JUnit
- Copy JUnit repository and paste in pom.xml

```

<project ...>
<groupId>softuni</groupId>
<artifactId>junit-example</artifactId>
<version>1.0-SNAPSHOT</version>
...
<properties>
    <maven.compiler.source> 13 </maven.compiler.source>
    <maven.compiler.target> 13 </maven.compiler.target>
</properties>

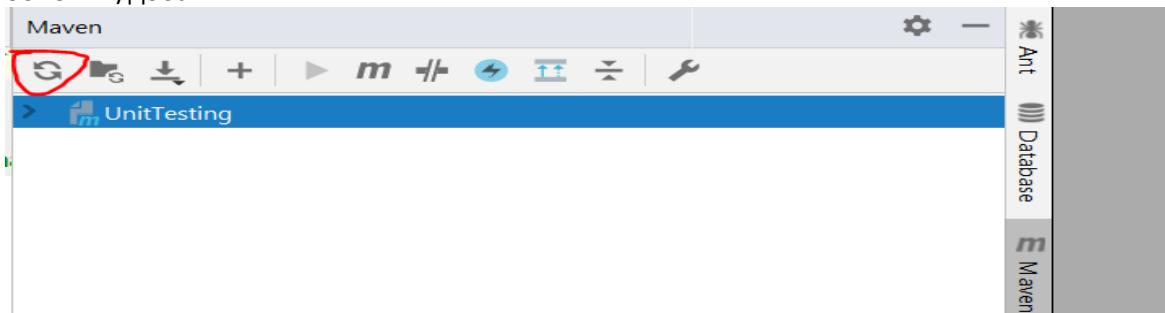
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>

        <scope>test</scope>
    </dependency>
</dependencies>

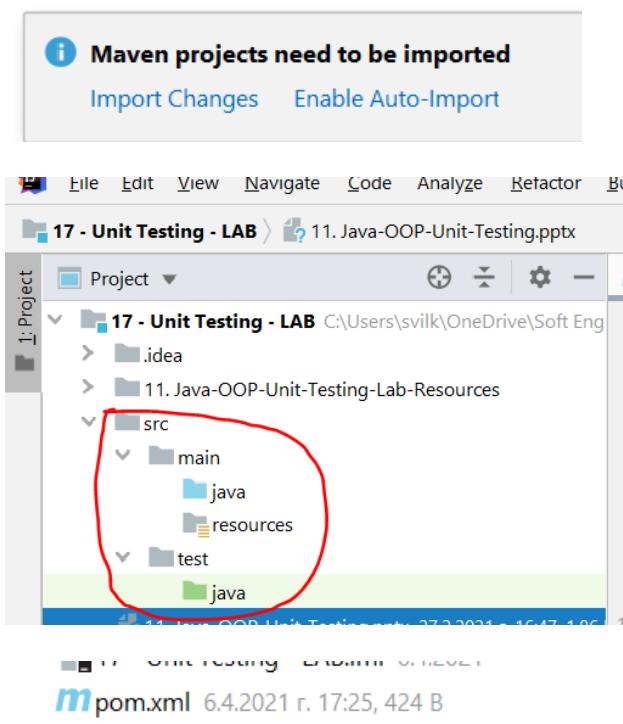
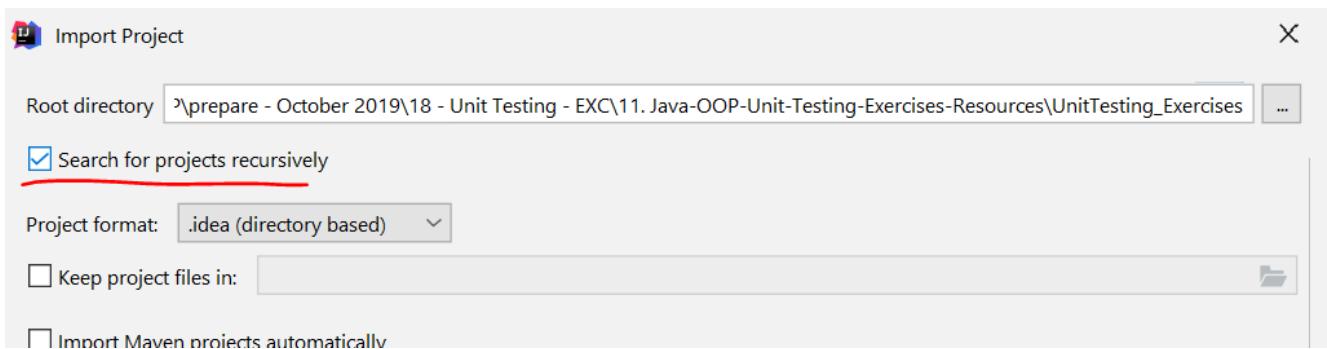
</project>

```

Re-import All Maven Projects – когато добавим ново dependency в pom.xml файла, от тук може да го обновим/добавим.

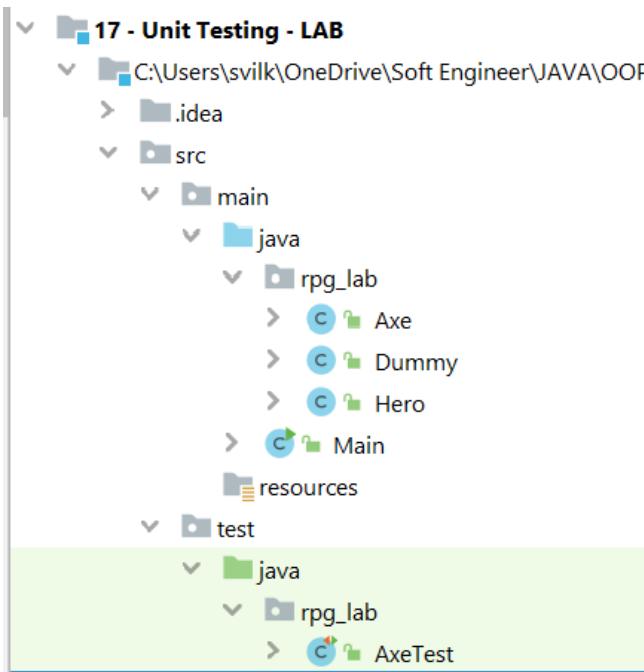


Намира проекти в директории надолу



В папката `test->java->rpg_lab` създаваме само класът, който ще тества AxeTest. И ако имаме други класове за тестване, то към края на името на класа добавяме Test.

И не копираме реалните класове от `main->java->rpg_lab` – те си стоят само в `main->java->rpg_lab`.



ВАЖНО: реално, имаме достъп до Junit както в **main->java**, така и в **test->java**.

Трябва да го използваме Junit само и единствено в папката **test->java**.

За да не допуснем грешка, за целта задаваме в pom.xml файла на Dependancy-то scope така:

<scope>test</scope>

Junit – Writing Tests

- Create new package (e.g. tests)
- Create a class for test methods (e.g. BankAccountTests)
- Create a public void method annotated with @Test

```
@Test
public void depositShouldAddMoney() {
    /* magic */
}
```

```
Държавна Анотация Test, част от junit
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Test {
    Class<? extends Throwable> expected() default Test.None.class;

    long timeout() default 0L;

    public static class None extends Throwable {
        private static final long serialVersionUID = 1L;

        private None() {
        }
    }
}
```

3A (mpri eў) Pattern

- **Arrange** – Preconditions – правим си мокитото примерно или си нагласяме обекта за тестване
- **Act** - Test a **single behavior** – пускаме обекта да работи

- **Assert** – Postconditions – сравняваме дали резултата от действието е това, което искаме

```
@Test
public void depositShouldAddMoney() {
    BankAccount account = new BankAccount();
    account.deposit(50);
    Assert.assertTrue(account.getBalance() == 50)
}
```

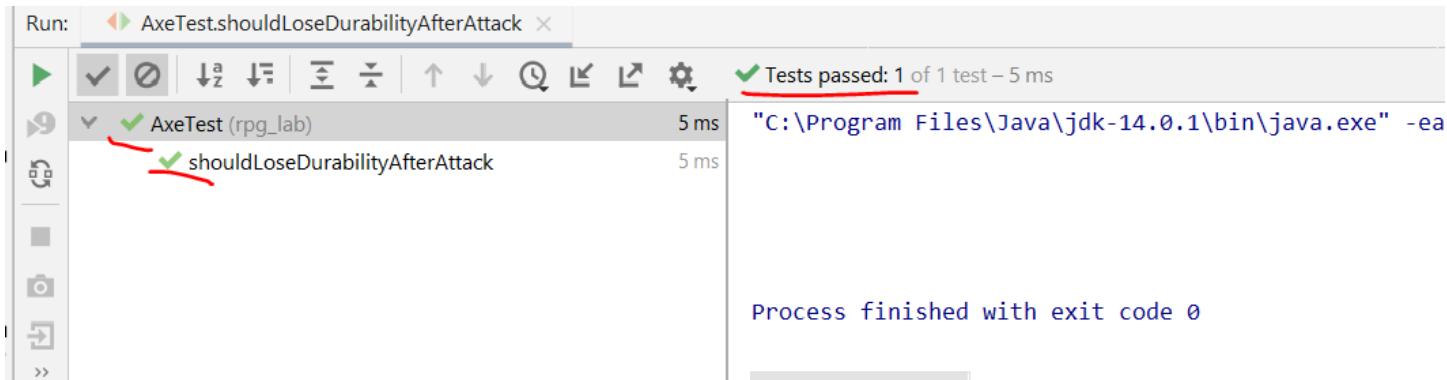
Exceptions – тук няма нужда да слагаме Assert

- Sometimes throwing an exception is the **expected behavior**

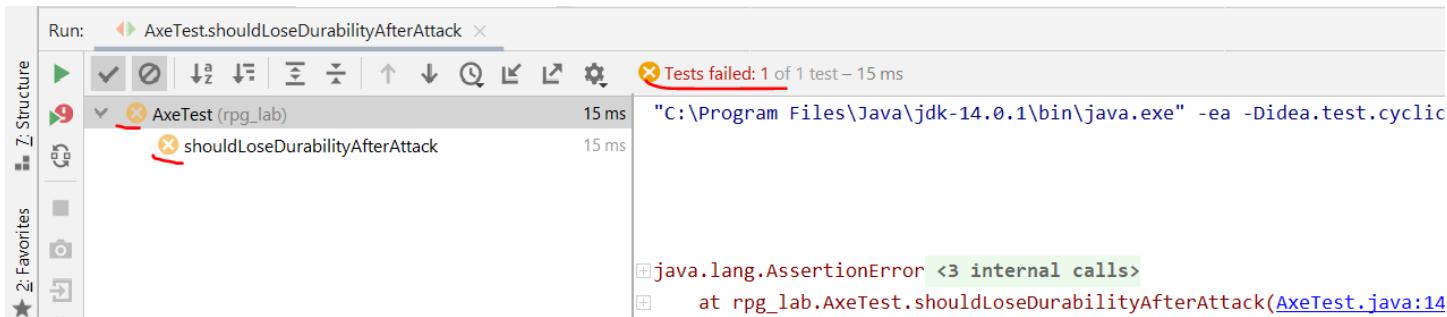
```
@Test(expected = IllegalArgumentException.class) //Assert
public void depositNegativeShouldNotAddMoney() {
    BankAccount account = new BankAccount(); //Arrange
    account.deposit(-50); //Act
}

Assert.assertTrue(axe.getDurabilityPoints() == 4);
```

Когато теста минава:



Когато теста не минава:



ВАЖНО – в един метод за тестване, тестваме само един единствен граничен случай.

Ако тестваме и четирите случая, то като гръмне теста, няма да знаем къде точно е грешката.

- Create the following tests
 - Dummy **loses health** if attacked
 - Dead Dummy **throws exception** if attacked
 - Dead Dummy **can give XP**
 - Alive Dummy **can't give XP**

```
import org.junit.Assert;
import org.junit.Test;
```

```

public class DummyTest {
    @Test
    public void dummyShouldLoseHealthWhenAttacked(){
        Dummy dummy = new Dummy(10, 10);
        dummy.takeAttack(5);
        Assert.assertTrue(dummy.getHealth() == 5);
    }

    @Test (expected = IllegalStateException.class)
    public void shouldThrowExceptionWhenAttackingDeadDummy(){
        Dummy dummy = new Dummy(-10, 10);
        dummy.takeAttack(10);
    }

    @Test
    public void dummyShouldGiveExperienceIfDead(){
        Dummy dummy = new Dummy(-10, 10);
        int actualExperience = dummy.giveExperience();
        Assert.assertTrue(actualExperience == 10);
    }

    @Test (expected = IllegalStateException.class)
    public void shouldThrowExceptionWhenGivingExperienceIfAlive(){
        Dummy dummy = new Dummy(10, 10);
        dummy.giveExperience();
    }
}

```

Вариант с `import static org.junit.Assert.*;`

```

import org.junit.Assert;
import org.junit.Test;
import static org.junit.Assert.*;

public class DummyTest {
    @Test
    public void dummyShouldLoseHealthWhenAttacked(){
        Dummy dummy = new Dummy(10, 10);
        dummy.takeAttack(5);
        assertTrue(dummy.getHealth() == 5);
    }
}

```

ВАЖНО: Тестовете ми не трябва да променят бизнес логиката и не трябва да добавят (нови) методи. Не трябва да пипам нивото на достъп(access modifiers). Тестовете ми не трябва да афектират кода като структура.

Ако имаме `private` методи, то трябва да се замислим дали има нужда да тестваме точно тях. По-добре да тестваме `public` методи (до които Junit има достъп), които извикват `private` методите.

32.4. Unit Testing Best Practices

Assertions

- `assertTrue()` vs `assertEquals()`
 - `assertTrue()`

```
Assert.assertTrue(account.getBalance() == 50);
```

`java.lang.AssertionError <3 internal calls>`

- `assertEquals(expected, actual)`

```
Assert.assertEquals(50, account.getBalance());
```

Better description when expecting value

```
java.lang.AssertionError:  
Expected :50  
Actual   :35  
<Click to see difference>
```

Assertion Messages

- Assertions can **show messages** – добавяне на съобщение
 - Helps with **diagnostics**

```
Assert.assertEquals("Wrong balance", 50, account.getBalance());
```

Helps finding the problem

```
java.lang.AssertionError: Wrong balance  
Expected :50  
Actual   :35  
<Click to see difference>
```

Magic Numbers – ако пишем директно числа вместо да използваме константи

- Avoid using magic numbers (use **constants** instead) – важи и за по принцип това правило

```
private static final int AMOUNT = 50;  
@Test  
public void depositShouldAddMoney() {  
    BankAccount account = new BankAccount();  
    account.deposit(AMOUNT);  
    Assert.assertEquals("Wrong balance", AMOUNT, account.getBalance(), delta 0.00);  
}
```

Use @Before annotation

```
private BankAccount account;  
@Before – изпълнява се преди започне всеки от тестовете  
@BeforeClass – изпълнява се преди започване изпълнението на тестовия клас  
@After – изпълнява се след като приключи изпълнението на тест класа, например за  
изчистване/зануляване на някакви променливи  
public void createAccount() {  
    this.account = new BankAccount();  
}  
@Test  
public void depositShouldAddMoney() { ... }
```

Или

```
private static final int BASE_ATTACK = 50;  
private static final int BASE_DURABILITY = 1;  
private static final int BASE_HEALTH = 10;  
private static final int BASE_EXPERIENCE = 10;  
private Dummy dummy;  
private Axe axe;  
  
@Before  
public void beforeEach() {  
    this.axe = new Axe(BASE_ATTACK, BASE_DURABILITY);  
    this.dummy = new Dummy(BASE_HEALTH, BASE_EXPERIENCE);  
}
```

Naming Test Methods

- Test names
 - Should use **business domain terminology**
 - Should be **descriptive and readable**

```
incrementNumber() {}
test1() {}
testTransfer() {}
```



```
depositAddsMoneyToBalance() {}
depositNegativeShouldNotAddMoney() {}
transferSubtractsFromSourceAddsToDestAccount() {}
```



We can debug the Junit test classes

32.5. Dependencies

32.5.1. Dependency Injection through constructor = създаваме interface/s

- Decouples classes and **makes code testable** – we use **Композиция/Composition here**

We want to test a **single behavior**

```
interface AccountManager { //Using interface
    Account getAccount();
}
```

```
public class Bank {
    private AccountManager accountManager; //Independent from Implementation

    public Bank(AccountManager accountManager) { // Injecting dependencies – банката използва
        accountManager чрез конструктора си
        this.accountManager = accountManager;
    }
}
```

32.5.2. Goal: Isolating Test Behavior

- In other words, to **fixate all moving parts**

```
@Test
public void testGetInfoById() {
    // Arrange
    AccountManager manager = new AccountManager() {
        public Account getAccount(String id) {..some code..} //имплементираме чрез
    Композиция/Composition метода, а не чрез Наследяване/Inheritance
    }
    Bank bank = new Bank(manager);

    // Act
    AccountInfo info = bank.getInfo(ID);

    // Assert...
    //Do some test here
}
```

32.5.3. Fake Implementations – when using big interfaces, for example when using database

- Not **readable**, cumbersome and boilerplate

```

@Test
public void testRequiresFakeImplementationOfBigInterface() {
    // Arrange
    Database db = new BankDatabase() {
        // Too many methods...
    };
    AccountManager manager = new AccountManager(db);
    // Act & Assert...
}

```

32.5.4. Mocking

32.6. Mocking

Когато искаме даден метод да ни връща винаги определена стойност

Когато нямаме конкретика в данните (примерно с Random), използваме Mocking – иначе два пъти работи теста ни, на третия път заради рандома не работи теста ни.

Може да си настроим да изпълнява/роверява/тества само определен метод от много методи и то само при определени условия – използва Reflection!!! – прави фалшива инстанция на класа, за който тестваме

- Mock objects **simulate behavior** of real objects
 - Supplies data exclusively for the test - e.g. network data, random data, big data (database), etc.

Mockist	VS.	Classical
<ul style="list-style-type: none"> • use fake objects • test behavior and state • need to know implementation details when writing tests • a bug in one class causes only tests for this class to fail • lower upfront cost of writing tests, but more changes might be needed in the future 		<ul style="list-style-type: none"> • use real objects as much as possible • test state only • don't care for implementation details when writing tests • a bug in one class causes a ripple in all of its consumers • higher upfront cost of writing unit tests, less changes in the future

Mockito – a framework for mocking objects

```

@Test
public void testAlarmClockShouldRingInTheMorning() {
    Time mockedTime = Mockito.mock(Time.class); //създадена фалшива инстанция на Time класа
    Mockito.when(mockedTime.isMorning()).thenReturn(true); //когато по-надолу ти се извика метод
    isMorning(), то върни true

```

```

    Mockito.when(sensor.popNextPressurePsiValue()).thenThrow(IllegalArgumentException.class);
    //когато по-надолу ти се извика метод popNextPressurePsiValue(), то върни exception

```

```

AlarmClock clock = new AlarmClock(mockedTime);
if (mockedTime.isMorning()) { //реално винаги е true
    Assert.assertTrue(clock.isRinging());
}

```

- Mockito Web Site - <https://site.mockito.org/>
- Mockito 3.0.0 dependency - <https://mvnrepository.com/artifact/org.mockito/mockito-core/3.0.0>
- Copy dependency in pom.xml

Изтегляме от сайта на Maven добавката Mockito

```
<!-- https://mvnrepository.com/artifact/org.mockito/mockito-core -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.9.0</version>
    <scope>test</scope>
</dependency>
```

Един от основните класове изглежда така:

```
public class Hero {
    private String name;
    private int experience;
    private Weapon weapon;
    private List<Weapon> inventory;
.......
```

Ако го правих с чист Reflection, то ще имаме 20 реда вместо 2 реда примерно

```
public class HeroTest {
    @Test
    public void shouldReceiveLootAfterKillingTarget(){
        Axe mockAxe = Mockito.mock(Axe.class); //няма имплементация, само обекта без методите му – тук мокваме клас Axe, който има interface Weapon
```

```
Mockito.when(mockAxe.getAttackPoints()).thenReturn(10); //когато викаме метода добавяме метода getAttackPoints() да връща стойност 10
```

```
Mockito.when(mockTarget.isDead()).thenReturn(true); //когато викаме метода isDead(), връщай винаги true
```

```
Target mockTarget = Mockito.mock(Target.class); //тук мокваме interface Target имплементиран в класа Dummy
Mockito.when(mockTarget.getLoot()).thenReturn(mockAxe);
Mockito.when(mockTarget.isDead()).thenReturn(true);
```

```
Hero hero = new Hero("asd", 0, mockAxe);
}
}
```

Когато използваме mocking, и за целите на тестването Композицията е за предпочтение пред Наследяването!!!

32.7. Mocking with Annotations

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.mock;

@ExtendWith(MockitoExtension.class) //JUnit5
@RunWith(MockitoJUnitRunner.class) //JUnit4
public class StudentSystemTest {
```

```

@Mock
private StudentParser parser;

same as this
    private StudentParser parser = mock(StudentParser.class);

@Mock
private StudentDatabase database;

@InjectMocks
private StudentSystem studentSystem;

@Test
public void createStudent_sampleInput_expectedResult() {
    String input = "Pesho,18";
    String expectedResult = "1,Pesho,18";
    Student student = new Student("Pesho", 18);
    Student withId = new Student(1, "Pesho", 18);

    when(parser.parseStudent(input)).thenReturn(student);
    when(database.persist(student)).thenReturn(withId);
    when(parser.formatStudent(withId)).thenReturn(expectedResult);

    String result = studentSystem.createStudent(input);

    assertEquals(expectedResult, result);
}

@Test
public void getStudentByFacultyNumberTest() {
    int facultyNumber = 10;
    Student student = new Student(10, "Georgi", 23);
    String formattedStudent = "10,Georgi,23";
    when(database.getStudentByFacultyNumber(facultyNumber)).thenReturn(student);
    when(parser.formatStudent(student)).thenReturn(formattedStudent);

    String result = studentSystem.getStudent(facultyNumber);

    assertEquals("10,Georgi,23", result);
}
}

```

32.8. Other libraries in Maven

Досега видяхме Junit и Mocking.

Hamcrest – библиотека за Matcher – <http://hamcrest.org/>
По-описателна

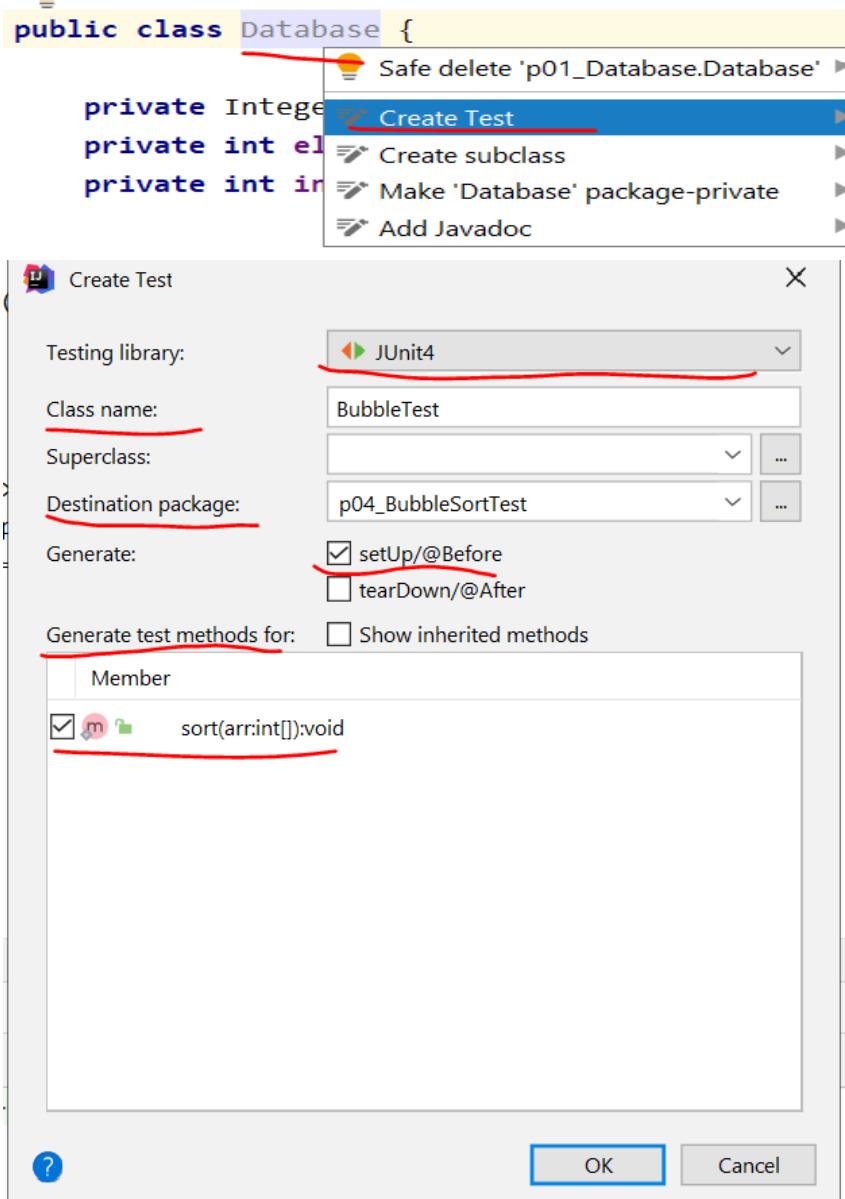
```

import static org.hamcrest.MatcherAssert.assertThat;
assertThat(theBiscuit, equalTo(myBiscuit));
MatcherAssert.assertThat(hero.getInventory(), containsAllOf(mockAxe)); - дали тази брадва я има в
инвентара

```

32.9. Още важни неща

Лесно автоматично създаване на тест класове - Alt + Enter – създава ги в съответните папки



От позитивни към негативни тестове

При Junit, всеки наш метод започва да тества при първоначални данни, или с други думи ако сме добавяли елемент в някаква структура чрез предходен тестов метод, то в текущия тестов метод все едно не сме добавяли/изменяли элемента все още.

Но, трябва да се внимава когато използваме **static** и **final** полета от масиви/колекции. Затова може в @Before да си ги добавяме като инициализация. Но Junit предпазва от това!!! 😊

При UnitTest, по-добре да добавяме exception към метода отколкото да използваме try/catch

Проверка дали два масива са еднакви с еднаква подредба – дългия вариант

```
@Test  
public void databaseCreationTestShouldSetElementsInCorrectOrderAccordingToInitialParameters() {  
    Integer[] elements = this.database.getElements();
```

```

boolean areEqual = true;
if (elements.length == numbers.length) {
    for (int i = 0; i < elements.length; i++) {
        if (!elements[i].equals(numbers[i])) {
            areEqual = false;
            break;
        }
    }
} else {
    areEqual = false;
}

Assert.assertTrue(areEqual);
}

```

Проверка дали два масива са еднакви с еднаква подредба – къс вариант

```

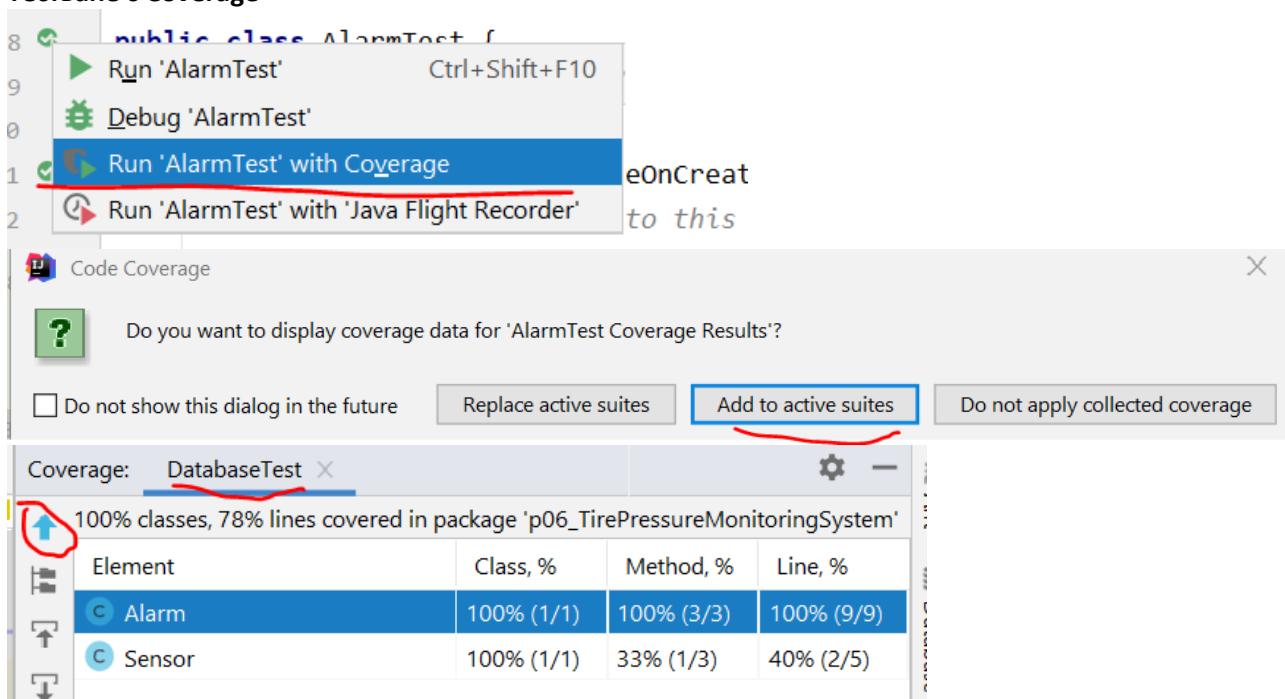
@Test
public void databaseCreationTestShouldSetElementsInCorrectOrderAccordingToInitialParameters() {
    Integer[] elements = this.database.getElements();

    Assert.assertArrayEquals(numbers, elements);
}

assertNotNull(field);
assertFalse(alarm.getAlarmOn());

```

Тестване с Coverage



```
66 List<Person> people = ne
67
68     if (username == null) {
69         throw new OperationInvali
70     }
71
72     for (Person person : ele
73         if (person == null)
74             continue;
75
76         if (person.getUsername()
77             people.add(person);
78     }
79
80 }
81
82 if (people.size() != 1)
83     throw new OperationInvali
84 }
85
86 return people.get(0);
87 }
```

Всички по-сложни колекции може да сведем до проверка на масиви – `Assert.assertArrayEqual()`;
Можем да сравняваме и списъци – `Assert.assertEquals()`; - така че няма проблем

Какво проверяваме обикновено:

От негативните:

- създаване на конструктора с null
- създаване на конструктор с повече елементи от разрешеното
- добавяне на некоректен елемент – например с дублирана или с отрицателна стойност
- добавяне на null element
- добавяне на коректен елемент извън размера/извън разрешения брой
- remove-не на елемент от празен списък/структура от данни – ръчно зануляваме структурата
- връща празна колекция – когато търсим елемент или правим друго нещо – извън границите на колекцията

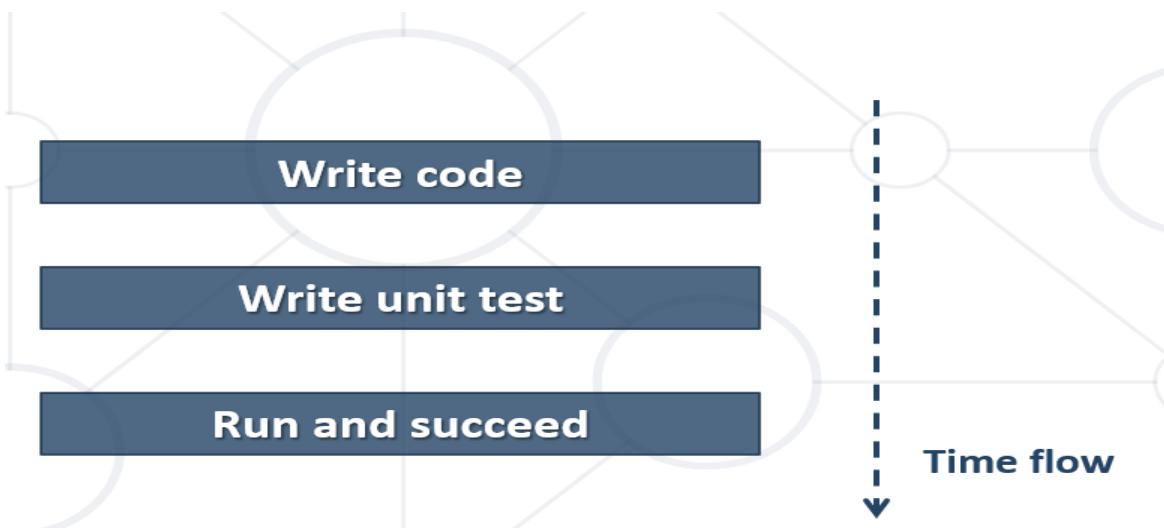
От позитивните:

- конструктора създава с валидни параметри
- добавяме на коректен елемент – сравняваме последно добавения
- нормално премахване на елемент – предпоследния е последния примерно
- връща сортирана колекция - когато търсим елемент или правим друго нещо – точно на границата на колекцията, и също така по средата на колекцията примерно

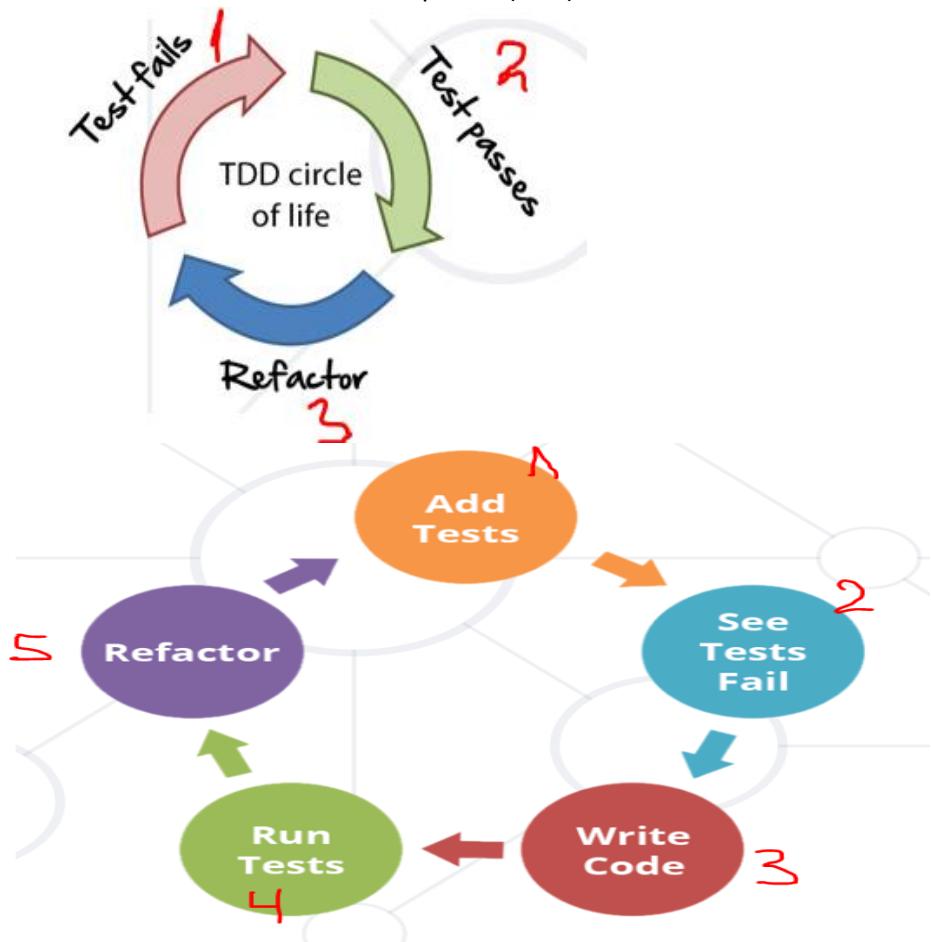
33. Test-Driven Development

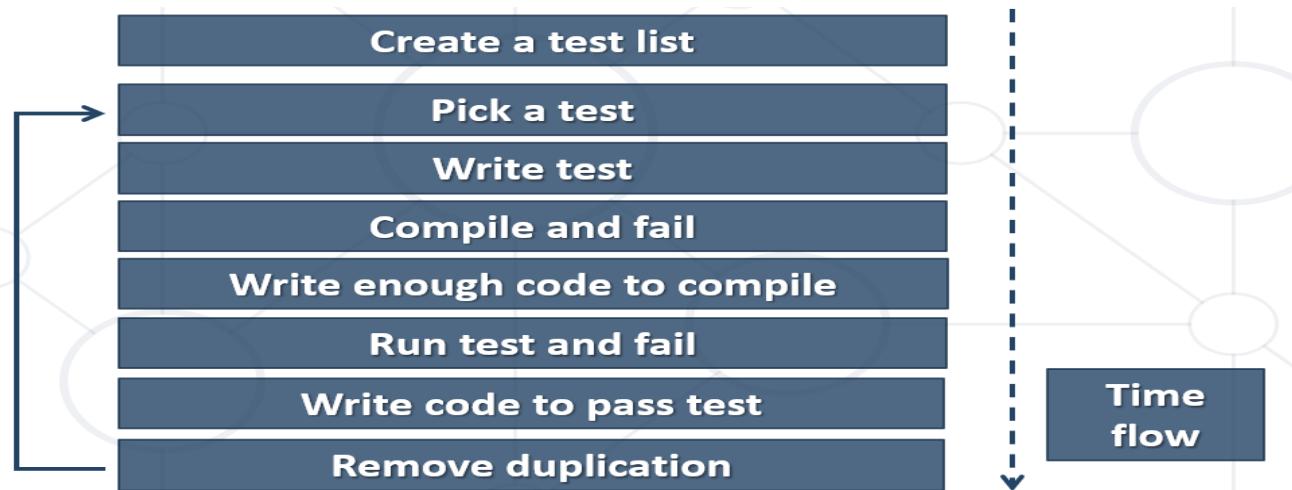
Unit Testing Approaches

- "Code First" (code and test) approach
 - Classical approach - Write code, then test it



- "Test First" approach
 - Test-driven development (TDD) - **Write tests first**





Why TDD?

- TDD helps **find design issues** early
 - Avoids reworking
- Writing code to satisfy a test is a **focused activity**
 - Less chance of an error
- Tests will be **more comprehensive** than if they are written after the code

34. Design Patterns

34.1. General Info

- **General** and **reusable solutions** to common problems in software design
- A **template** for solving given problems
- Add additional layers of **abstraction** in order to reach flexibility
- Patterns solve **software structural problems** like:
 - Abstraction
 - Encapsulation
 - Separation of concerns
 - Coupling and cohesion
 - Separation of interface and implementation
 - Divide and conquer

Elements of a Design Pattern

- Pattern name - Increases **vocabulary** of designers
- Problem - **Intent**, context and when to apply
- Solution - **Abstract** code
- Consequences - **Results** and trade-offs

Why Design Patterns?

Benefits

- Names form a common vocabulary
- Enable large-scale **reuse** of software architectures
- Help improve developer **communication**
- Help ease the **transition** to Object Oriented technology
- Can **speed-up** the development

Drawbacks

- Do not lead to a direct code reuse
- Deceptively simple
- Developers may suffer from **pattern overload** and **overdesign**
- Validated by **experience** and discussion, not by automated testing
- Should be used only if **understood well**

34.2. Types of Design Patterns

34.2.1. *Creational patterns*

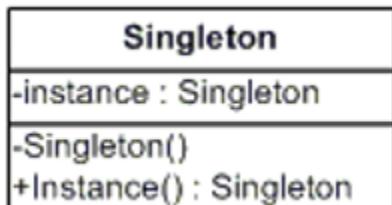
- Deal with **initialization and configuration** of classes and objects

Purposes

- Deal with **object creation** mechanisms
- Trying to create objects in a **manner suitable** to the **situation**
- Two main ideas
 - **Encapsulating** knowledge about which classes the system uses
 - **Hiding** how instances of these classes are created

A. Singleton Pattern

- The most often used creational design pattern
- A Singleton class is supposed to have **only one instance**
- It is **not a global variable**
- Possible problems
 - Lazy loading – когато ни потрябва, тогава да инициализираме
 - Thread-safe – например, две нишки по едно и също време ако има –
 - Корато ни потрябва, да не инициализираме нова инстанция, а да връщаме същата



```
class SerializableSingleton {
    private static SerializableSingleton instance;

    private SerializableSingleton() {}

    public static synchronized SerializableSingleton getInstance() {
        if(instance == null) {
            instance = new SerializableSingleton();
        }
        return instance;
    }
}
```

Пример:

```

import java.util.HashMap;
import java.util.Map;

public class SingletonDataContainer implements SingletonContainer {
    private static SingletonDataContainer instance;
    private Map<String, Integer> capitals;

    private SingletonDataContainer() {
        this.capitals = new HashMap<>();
        System.out.println("Initializing singleton object");
    }

    public int getPopulation(Map<String, Integer> capitals, String name) { return capitals.get(name); }

    public int getPopulation(Map<String, Integer> capitals, String name) { return capitals.get(name); }

    public static SingletonDataContainer getInstance() {
        if (instance != null){
            return instance;
        }
        instance = new SingletonDataContainer();
        return instance;
    }

import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> capitals = new HashMap<>();

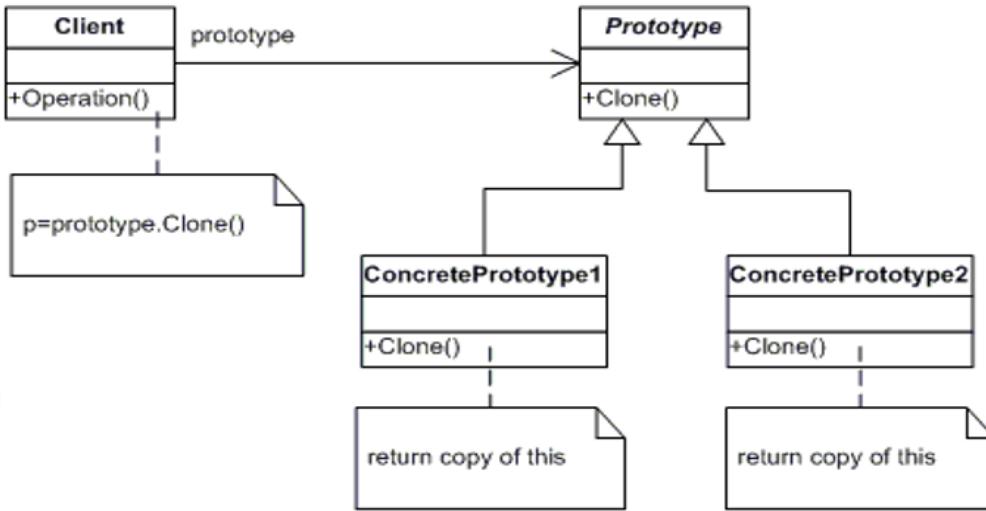
        capitals.put("Sofia", 120000);
        capitals.put("Varna", 90000);

        SingletonDataContainer instance = SingletonDataContainer.getInstance();
        System.out.println(instance.getPopulation(capitals, name: "Sofia"));
        SingletonDataContainer instance1 = SingletonDataContainer.getInstance();
        System.out.println(instance1.getPopulation(capitals, name: "Varna"));
    }
}

```

B. Prototype Pattern – създаване на обекти чрез възможността да ги клонираме/копираме

- Factory for **cloning** new instances from a prototype
 - Create new objects by copying this prototype
 - Instead if using the "new" keyword
- **Cloneable** interface acts as Prototype



Пример за точка с координати x и y.

```

public class Main {
    public static void main(String[] args) {
        Point2D a = new Point2D(3, 4);

        Point2D b = a.clone();
    }
}

public class Point2D {
    private int x;
    private int y;

    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point2D clone(){
        return new Point2D(x, y);
    }
}

```

C. Builder Pattern – стъпка по стъпка създава обекта като имаме контрол на всяка стъпка

Не създаваме конструктори с различен брой параметри, а правим chain-ване на седъри.

- Separates the construction of a complex object from its representation
 - Same construction process can create different representations
- Provides control over steps of construction process

Класически пример:

```

public class CarBuilder {
    private String type;
    private String color;
    private int numberOfDoors;
    private String city;
    private String address;

    public CarBuilder() {
    }
}

```

```

public CarBuilder withType(String type) {
    this.type = type;
    return this; //връща текущата инстанция на класа, за да може да има chain-ване при
създаване на обекта
}

public CarBuilder withColor(String color) {
    this.color = color;
    return this; //връща текущата инстанция на класа, за да може да има chain-ване при
създаване на обекта
}

public CarBuilder withCity(String city) {
    this.city = city;
    return this; //връща текущата инстанция на класа, за да може да има chain-ване при
създаване на обекта
}

public CarBuilder withAddress(String address) {
    this.city = city;
    return this; //връща текущата инстанция на класа, за да може да има chain-ване при
създаване на обекта
}

public CarBuilder withNumberOfDoors(int numberOfDoors) {
    this.numberOfDoors = numberOfDoors;
    return this; //връща текущата инстанция на класа, за да може да има chain-ване при
създаване на обекта
}

public Car build() {
    Car car = new Car();
    if (this.type != null) {
        car.setType(this.type);
    }

    if (this.color != null) {
        car.setColor(this.color);
    }

    if (this.numberOfDoors != 0) {
        car.setNumberOfDoors(this.numberOfDoors);
    }

    if (this.city != null) {
        car.setCity(this.city);
    }

    if (this.address != null) {
        car.setAddress(this.address);
    }

    return car; //връща целият новосъздаден обект на база нещата, които сме вкарали в
конструктора за този обект – някои данни сме вкарали, други сме пропуснали.
}
}

public class Car {
    private String type;
    private String color;

```

```
private int numberOfDoors;
private String city;
private String address;

public String getType() {
    return this.type;
}

public void setType(String type) {
    this.type = type;
}

public String getColor() {
    return this.color;
}

public void setColor(String color) {
    this.color = color;
}

public int getNumberOfDoors() {
    return this.numberOfDoors;
}

public void setNumberOfDoors(int numberOfDoors) {
    this.numberOfDoors = numberOfDoors;
}

public String getCity() {
    return this.city;
}

public void setCity(String city) {
    this.city = city;
}

public String getAddress() {
    return this.address;
}

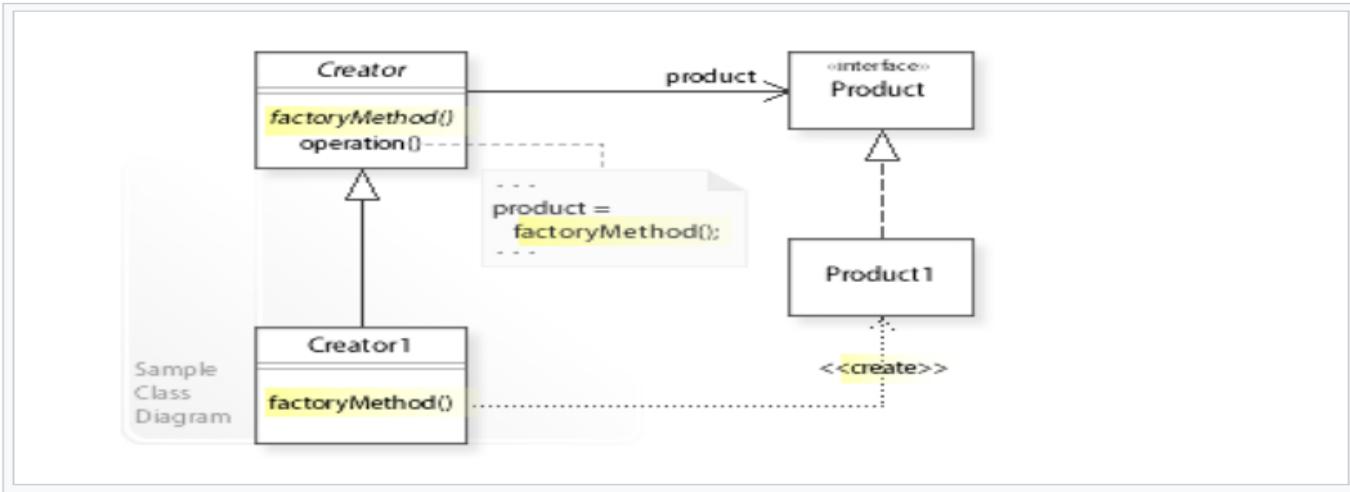
public void setAddress(String address) {
    this.address = address;
}

public class Main {
    public static void main(String[] args) {

        //Chain-ване
        Car car = new CarBuilder()
            .withType("Sedan") //Връща текущата инстанция на класа
            .withType("Cabrio") //запазва последното
            .withColor("Purple") //Връща текущата инстанция на класа
            .withNumberOfDoors(5) //Връща текущата инстанция на класа
            .build();
    }
}
```

D. Factory pattern

UML class diagram [edit]



A sample UML class diagram for the Factory Method design pattern. [4]

```

public interface Factory {
    Animal createAnimal(String type);
}
public class AnimalFactory implements Factory {
    private Forest forest;

    public AnimalFactory(Forest forest) {
        this.forest = forest; //инстанцията на това животно е в среда за обитаване гора forest
    }

    @Override
    public Animal createAnimal(String type) {
        Animal animal = null;

        switch (type) {
            case "Monkey":
                animal = new Monkey();
            case "ProgrammerAnimal":
                animal = new ProgrammerAnimal();
            case "Giraffe":
                animal = new Giraffe();
            case "Cat":
                animal = new Cat();
            default:
                animal = null;
        }

        if (animal != null) {
            this.forest.addAnimal(animal); //викаме на гората метода за добавяне на животно
        }
        return animal;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        Forest forest = new OakForest();
    }
}

```

```

Factory factory = new AnimalFactory(forest);

String type = sc.nextLine();
while (!type.equals("End")) {
    factory.createAnimal(type); //тук реално като добавяме животно, то се добавя и в
самата гора forest
    type = sc.nextLine();
}
}

public interface Animal {
    void makeSound();
}

public class Cat implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Miau");
    }
}

public class Giraffe implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Muuuu aaa");
    }
}

public class Monkey implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Uaaaa-haaaa");
    }
}

public interface Forest {
    void addAnimal(Animal animal);
}

public class OakForest implements Forest {
    private List<Animal> animals;

    public OakForest() {
        this.animals = new ArrayList<>();
    }

    @Override
    public void addAnimal(Animal animal) { //метод за добавяне на животно в гората
        this.animals.add(animal);
    }
}

```

34.2.2. Structural patterns

- Describe ways to **assemble** objects to implement **new functionality**
- **Composition** of classes and objects

Purposes

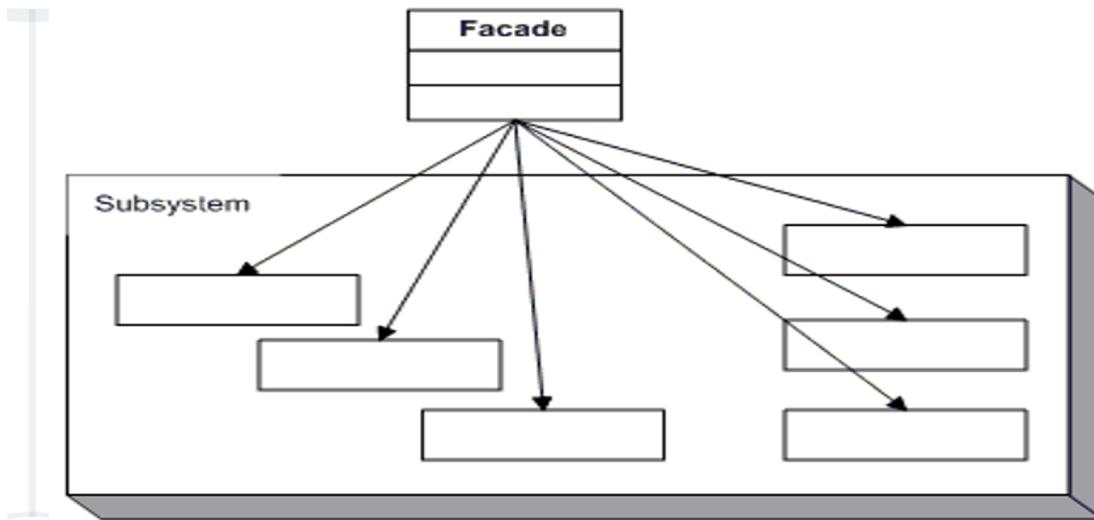
- Describe ways to assemble **objects** to implement a **new functionality**
- Ease the design by identifying a simple way to realize **relationship** between entities

- All about Class and Object composition
 - Inheritance to compose interfaces
 - Ways to compose objects to obtain **new functionality**

A. Façade Pattern – клиентът знае само за Façade интерфейса.

- Provides an **unified interface** to a set of interfaces in a subsystem
- Defines a **higher-level interface** that makes the subsystem easier to use

Когато имаме стари системи или се сменя софтуера, и за да работи новия софтуер или 3d party нещата, и то без да се пише кода наново, то просто се прави фасада. User-а борави с фасадата, но фасадата може да взаимодейства вече с нов имплементация на нов софтуер/език.



Използват композиция и делегация.

The Façade Class

```

class Facade {
    private SubSystemOne one;
    private SubSystemTwo two;

    public Facade() {
        one = new SubSystemOne();
        two = new SubSystemTwo();
    }
}

```

The Façade Class

```

public void MethodA() {
    System.out.println("MethodA() ---- ");
    one.MethodOne();
    two.MethodTwo();
}

public void MethodB() {
    System.out.println("MethodB() ---- ");
    two.MethodTwo(); }
}

```

Subsystem Classes

```

class SubSystemOne {
    public void MethodOne() {
        System.out.println(" SubSystemOne Method"); }
}

```

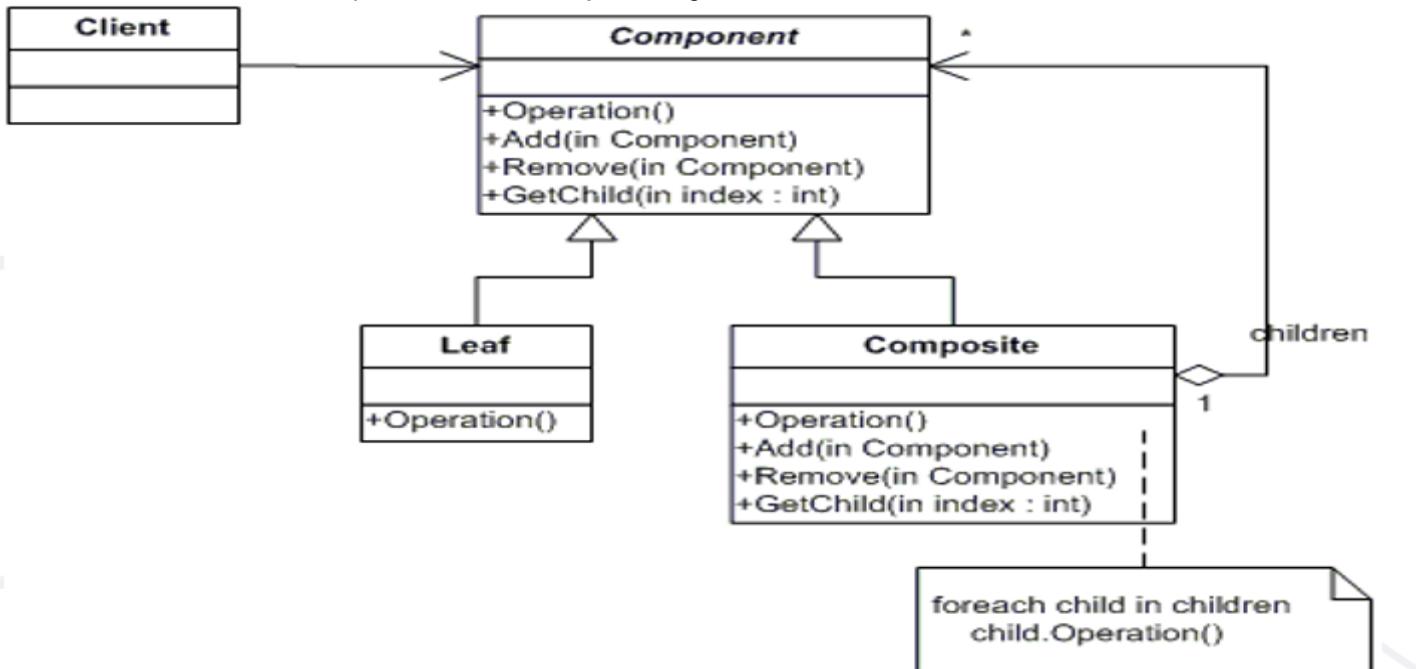
```

class SubSystemTwo {
    public void MethodTwo() {
        System.out.println(" SubSystemTwo Method");
    }
}

```

B. Composite Pattern – полиморфизъм с екстри

- Allows to **combine** different types of objects in **tree structures**
- Gives the possibility to treat the **same object(s)**
- Used when
 - You have different objects that you want to **treat the same way**
 - You want to present **hierarchy** of objects



The Component Abstract Class

```

abstract class Component {
    protected String name;

    public Component(String name) {
        this.name = name;
    }

    public abstract void add(Component c);
    public abstract void remove(Component c);
    public abstract void display(int depth);
}

```

The Composite Class

```

class Composite extends Component {
    private List<Component> children = new ArrayList<Component>();

    public Composite(String name) {
        super(name);
    }

    @Override
    public void add(Component component) {
        children.add(component);
    }
}

```

```

@Override
public void remove(Component component) {
    children.Remove(component);
}

@Override
public void display(int depth) {
    System.out.println(printNameInDepth(depth, name));
    foreach(Component component :children){
        component.display(depth + 2);
    }
}

public void printNameInDepth(int depth, String name) {
    for (int i = 0; i < depth; i++)
        System.out.print("-");
    System.out.print(name);
}

```

The Leaf Class

```

class Leaf extends Component {
    public Leaf(String name) {
        super(name);
    }

    @Override
    public void add(Component c) {
        System.out.println("Cannot add to a leaf");
    }

    @Override
    public void Remove(Component c) {
        System.out.println("Cannot remove from a leaf");
    }

    @Override
    public void Display(int depth) {
        System.out.println(printNameInDepth(depth, name));
    }
}

```

34.2.3. Behavioral patterns

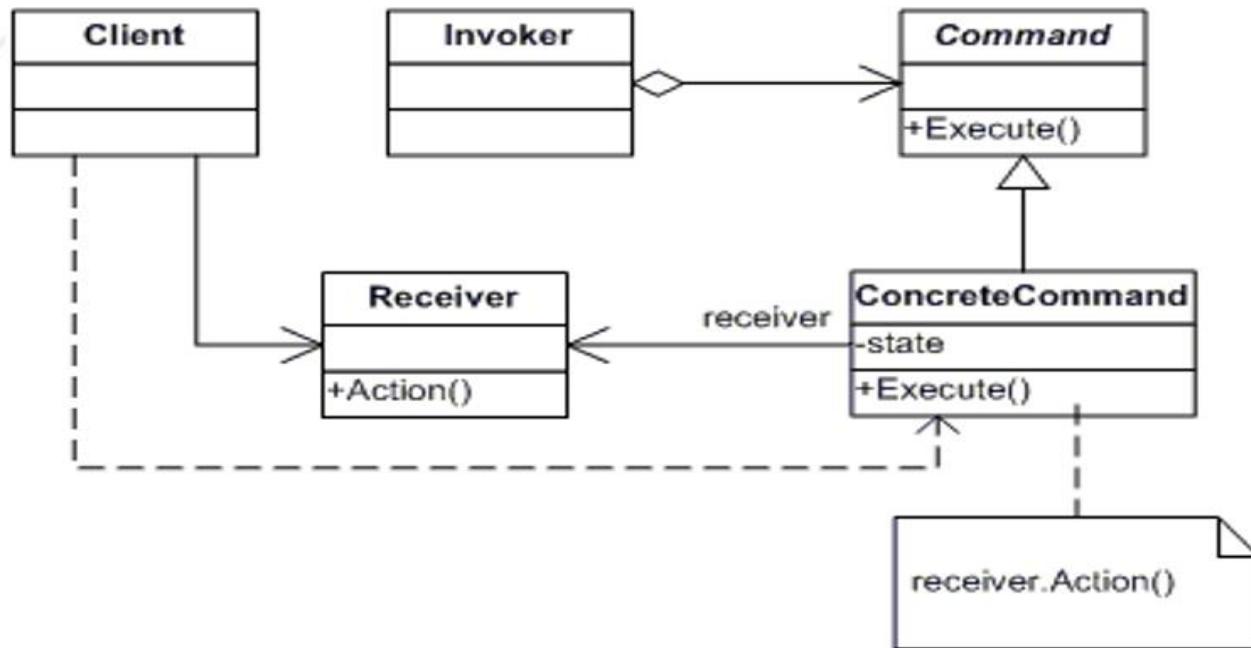
- Deal with dynamic **interactions** among societies of classes
- Distribute **responsibility**

Purposes

- Concerned with **interaction** between objects
 - Either with the **assignment of responsibilities** between objects
 - Or **encapsulating behavior** in an object and delegating requests to it
- Increases **flexibility** in carrying out cross-classes communication

A. Command Pattern

- An object **encapsulates** all the information needed to call a method at a later time
 - Let you **parameterize** clients with different requests, queue or log requests, and support undoable operations



The Command Abstract Class/Interface

```
public interface Command {  
    String execute();  
}
```

Concrete Command Classes

```
public class IncreaseProductPriceCommand implements Command{  
    private int amount;  
    private Product product;  
  
    public IncreaseProductPriceCommand (Product product, int amount) {  
        this.product = product;  
        this.amount = amount;  
    }  
  
    @Override  
    public String execute() {  
        this.product.increasePrice(this.amount);  
  
        return String.format("The price for %s has been increased by %d",  
            this.product.getName(), this.amount);  
    }  
}  
  
public class DecreaseProductPriceCommand implements Command {  
    private Product product;  
    private int amount;  
  
    public DecreaseProductPriceCommand(Product product, int amount) {  
        this.product = product;  
        this.amount = amount;  
    }
```

```

@Override
public String execute() {
    this.product.increasePrice(this.amount);

    return String.format("The price for %s has been decreased by %d",
        this.product.getName(), this.amount);
}
}

```

The Receiver Class - Command design pattern states that we shouldn't use receiver classes directly:

```

public class Product {
    private String name;
    private int price;

    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPrice() {
        return this.price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public void increasePrice(int amount) {
        this.setPrice(this.price + amount);
    }

    public void decreasePrice(int amount) {
        this.setPrice(this.price - amount);
    }
}

```

The Invoker Class

```

public class ModifyPrice {
    private List<Command> commands;

    public ModifyPrice() {
        this.commands = new ArrayList<>();
    }

    public void addCommand(Command command) {
        this.commands.add(command);
    }
}

```

```

    public void invoke() {
        this.commands.forEach(c -> System.out.println(c.execute()));
    }
}

```

Main класа:

```

public class Main {
    public static void main(String[] args) {
        Product product = new Product("Phone", 500);
        ModifyPrice modifier = new ModifyPrice();

        modifier.addCommand(new IncreaseProductPriceCommand(product, 100));
        modifier.addCommand(new DecreaseProductPriceCommand(product, 20));
        modifier.addCommand(new DecreaseProductPriceCommand(product, 45));

        modifier.invoke();

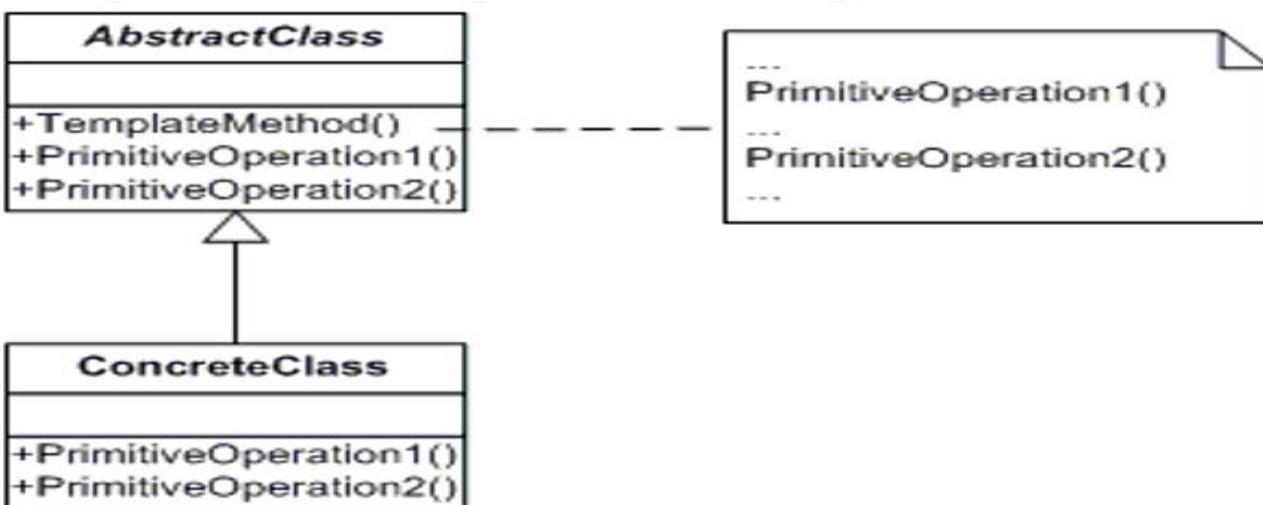
        System.out.println(product.getPrice());
    }
}

```

B. Template Pattern

Дефинираме шаблон/template за някакъв алгоритъм, и оставяме имплементацията на някой друг. Наследяване и полиморфизъм прилагаме.

- Define the **skeleton** of an algorithm in a method, leaving some implementation to its subclasses
- Allows the subclasses to **redefine** the implementation of some of the **parts** of the algorithm, but not its structure



The Abstract Class

```

abstract class AbstractClass {
    public abstract void primitiveOperation1();
    public abstract void primitiveOperation2();

    public void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        System.out.println("");
    }
}

```

A Concrete Class

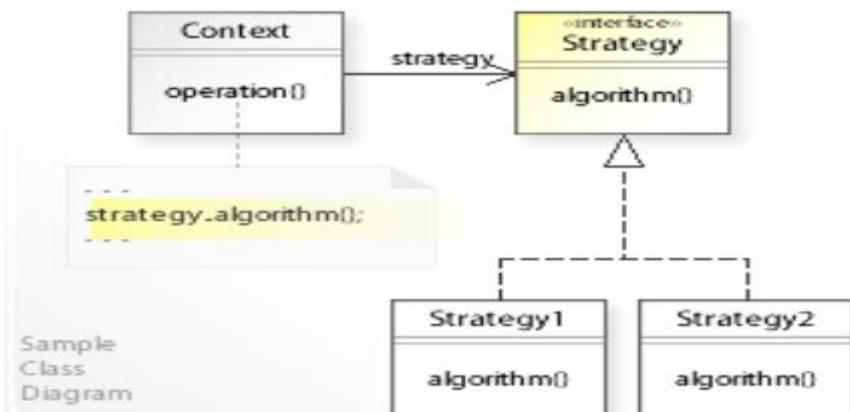
```
class ConcreteClassA extends AbstractClass {  
    @Override  
    public void primitiveOperation1() {  
        System.out.println("ConcreteClassA.primitiveOperation1()");  
    }  
  
    @Override  
    public void primitiveOperation2() {  
        System.out.println("ConcreteClassA.primitiveOperation2()");  
    }  
}
```

C. Strategy Pattern

Данните се пазят в един клас, а изпълнението на този клас се пази в друг клас.

От отвън да инжектираме как да работи класа.

Един нов обект казва как да се държи един стар обект.



```
public interface EatBehaviour {  
    void eat();  
}  
public class MessyEatBehaviour implements EatBehaviour{  
    @Override  
    public void eat() {  
        System.out.println("I ate a lot and I made a mess!");  
    }  
}  
public class CleanEatBehaviour implements EatBehaviour {  
    @Override  
    public void eat() {  
        System.out.println("I ate some and everything is clean!");  
    }  
}  
public class NotHungryEatBehaviour implements EatBehaviour {  
    @Override  
    public void eat() {  
        System.out.println("I am not hungry - I am not a real cat!");  
    }  
}  
  
public class Cat {  
    private EatBehaviour eatBehaviour;  
  
    public Cat(EatBehaviour eatBehaviour) {  
        this.eatBehaviour = eatBehaviour;  
    }  
}
```

```

public void howToEat() {
    this.eatBehaviour.eat();
}
}

public class Main {
    public static void main(String[] args) {
        Cat cat1 = new Cat(new NotHungryEatBehaviour());
        cat1.howToEat();

        Cat cat2 = new Cat(new MessyEatBehaviour());
        cat2.howToEat();

        Cat cat3 = new Cat(new CleanEatBehaviour());
        cat3.howToEat();
    }
}

```

35. Wild cards

The question mark (?) is known as the wildcard in generic programming . It represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type. Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

Types of wildcards in Java:

1. Upper Bounded Wildcards:

These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < integer >, List < double >, and List < number > , you can do this using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its **upper bound (parent bound)**.

Syntax: public static void add(List<? extends Number> list)

```

//Java program to demonstrate Upper Bounded Wildcards
import java.util.Arrays;
import java.util.List;

class Test {
    public static void main(String[] args) {
        //Upper Bounded Integer List
        List<Integer> list1 = Arrays.asList(4, 5, 6, 7);

        //printing the sum of elements in List
        System.out.println("Total sum is:" + sum(list1));

        //Double List
        List<Double> list2 = Arrays.asList(4.1, 5.1, 6.1);

        //printing the sum of elements in List
        System.out.print("Total sum is:" + sum(list2));
    }
}

```

```

private static double sum(List<? extends Number> list) {
    double sum = 0.0;
    for (Number i : list) {
        sum += i.doubleValue();
    }

    return sum;
}
}

```

In the above program, list1 and list2 are objects of the List class. list1 is a collection of Integer and list2 is a collection of Double. Both of them are being passed to method sum which has a wildcard that extends Number. This means that list being passed can be of any field or subclass of that field. **Here, Integer and Double are subclasses of class Number.**

2. Lower Bounded Wildcards:

It is expressed using the wildcard character ('?'), followed by the super keyword, followed by its **lower (child) bound**: <? super A>.

Syntax: Collectiontype <? super A>

```

//Java program to demonstrate Lower Bounded Wildcards
import java.util.Arrays;
import java.util.List;

class Test {
    public static void main(String[] args) {
        //Lower Bounded Integer List
        List<Integer> list1 = Arrays.asList(4, 5, 6, 7);

        //Integer List object is being passed
        printOnlyIntegerClassorSuperClass(list1);

        //Number List
        List<Number> list2 = Arrays.asList(4, 5, 6, 7);

        //Integer List object is being passed
        printOnlyIntegerClassorSuperClass(list2);
    }

    public static void printOnlyIntegerClassorSuperClass(List<? super Integer> list) {
        System.out.println(list);
    }
}

```

Here arguments can be Integer or superclass of Integer(which is Number). The method printOnlyIntegerClassorSuperClass will only take Integer or its superclass objects. However if we pass list of type Double then we will get compilation error. It is because only the Integer field or its superclass can be passed . Double is not the superclass of Integer.

Use **extend wildcard** when **you want to get values out of a structure**

and **super wildcard** when **you put values in a structure**.

Don't use wildcard when you get and put values in a structure. Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

3. Unbounded Wildcard:

This wildcard type is specified using the wildcard character (?), for example, List. This is called a **list of unknown type**. These are useful in the following cases

- When writing a method which can be employed using functionality **provided in Object class**.
- When the **code** is using **methods** in the generic class **that don't depend on the type parameter**

```
//Java program to demonstrate Unbounded wildcard
import java.util.Arrays;
import java.util.List;

class Test {
    public static void main(String[] args) {
        //Integer List
        List<Integer> list1 = Arrays.asList(1, 2, 3);

        //Double List
        List<Double> list2 = Arrays.asList(1.1, 2.2, 3.3);

        printlist(list1);

        printlist(list2);
    }

    private static void printlist(List<?> list) {
        System.out.println(list);
    }
}
```

36. Automatic Garbage Collection

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

In Java, process of deallocating memory is handled automatically by the garbage collector.

Ако променливата е поле на класа, се събира чак когато класа стане готов за събиране.

Ако е локална променлива – чак след края на блока.

37. Нишки и влакна – Threads and Fibers

37.1. Parallel programming

threads and fibers – нишки и влакна – ниско ниво код близко до hardware

```
public class Main implements Runnable {
    public static void main(String[] args) {
        new Main().run();
    }

    @Override
    public void run() {
        int n = 10;

        while (n-- > 0){
```

```

        Thread thread = new Thread(this::doSomething); //вземаме нишка от операционната
система / изключително скъпа операция
        thread.start();
    }

}

private void doSomething() {
    long z = 0;

    while (true){
        z++;
    }
}
}

```

Thread.sleep(1000);
System.out.println(Thread.currentThread()); - връща името на текущата нишка
System.out.println(ForkJoinPool.commonPool());

```

public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
    System.out.println(Runtime.getRuntime().availableProcessors()); - връща 1 ядро по-малко,
защото едно ядро е заето с нишката на main метода
    System.out.println(ForkJoinPool.commonPool()); - - връща състоянието на pool-а
}

```

Imperative style - обикновени цикли, if-else клаузи - как ще се случи
Functional/Declarative style - какво ще се случи (без значение как), не се интересуваме от самото изпълнение, а от логиката - .stream - този стил е много близък до многонишковото програмиране и е higher level of abstraction

```

public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    numbers.parallelStream()
        .map(e -> transform(e))
        .forEach(System.out::println);
}

```

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
numbers.stream().parallel(); равно на numbers.parallelStream(); - произволен ред ги печата
numbers.stream().sequential(); - в последователен ред ги печата.

In JAVA (Java 8) we do not have Reactive streaming.

Streams
sequential vs parallel

entire pipeline is
sequential or parallel
no segments

Reactive Stream
sync vs. async

Depends

subscribeOn - no segments
observeOn - segments

Прави ги парарелно, но ги печата подредени.

```
numbers.parallelStream()  
    .map(e -> transform(e))  
    .forEachOrdered(e -> printIt(e));
```

Примери с `.reduce()` метода: - като агрегатор

//Вземи сбора на всички числа

```
int[] numbers = new int[]{1, 2, 3, 4, 5};  
int sum = Arrays.stream(numbers).reduce(0, (val, num) -> val += num);
```

//вземи най-дългата дума

```
String[] words = new String[]{"hello", "pesho", "abc", "worlddd"};  
String longestWord = Arrays.stream(words).reduce("", (val, w) -> {  
    if (w.length() > val.length()) {  
        val = w;  
    }  
    return val;  
});
```

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);  
numbers.parallelStream().reduce(0, (total,e) -> add(total, e)); //0 е identity value  
private static Integer add(Integer total, Integer e) {  
    return total+= e;  
}
```

`//reduce does not take initial value, it takes identity value - it is the value which will not
change the operation if you apply it on`

X + identity value 0
Y * identity value 1

Да внимаваме - ако превключим от sequential към parallel, identity value трябва да е коректно зададено. Не е ок да напишем 36 години, при паралелното дава друг резултат.

The question is - How many threads should I create? How much food should I eat?
Ако съзdam прекалено много нишки, даже по-бавно би работило.

I0 intensive - прави паузи често, тогава може да имаме повече нишки

Computation intensive vs. IO intensive

For Computation intensive

Threads <= # of cores

For IO intensive

Threads may be greater than # of cores ????

of Cores
#T <= -----
1 - blocking factor

0 <= blocking factor < 1

Configuring number of threads JVM wide:

Djava.util.concurrent.ForkJoinPool.common.parallelism=100 100 нитки

37.2. Asynchronous programming

In JAVA it is CompletableFuture

In JS it is Promises

```
public static CompletableFuture<Integer> create() {  
    return CompletableFuture.supplyAsync(() -> 2);  
}
```

```
CompletableFuture<Integer> future = create();  
CompletableFuture<Void> future2 = future.thenAccept(data -> System.out.println(data));  
  
create()  
    .thenAccept(data -> System.out.println(data))  
    .thenApply(d -> d * 10)  
    .thenRun(() -> System.out.println("This never dies"));
```

	Stream	CompletableFuture
1	pipeline	pipeline
2	lazy	lazy
3	zero, one, or more data	zero or one
4	only data channel	data channel and error channel
5	forEach	thenAccept
6	map	thenApply
7	exception - <u>oops</u>	error channel

```
.thenAccept(data -> System.out.println(data))
Consumer<String> printer = str -> System.out.println(str);
Arrays.stream(sc.nextLine().split("\\s+")).forEach( e-> printer.accept(e));

.thenApply(d -> d * 10)
Function<Integer, Integer> squared = x -> x * x; - първият е входните данни, втория параметър е
типа на изхода
System.out.println(squared.apply(25)); //връща 625
```

```
.thenRun(() -> System.out.println("This never dies"));
Supplier<Integer> genRandomInt = () -> new Random().nextInt(51);
int rnd = genRandomInt.get();
```

```
Adding data to the pipeline
public static void main(String[] args) throws InterruptedException {
    CompletableFuture<Integer> future = new CompletableFuture<Integer>();
    future
        .thenApply(d -> d * 2)
        .thenApply(d -> d + 1)
        .thenAccept(data -> System.out.println(data));
    System.out.println("We built the pipeline");
    System.out.println("Prepared to print");

    sleep(1000);

    future.complete(2);
    sleep(1000);
}
```

Working with exceptions

```

public static void main(String[] args) {
    create()
        .thenApply(data -> data * 2)
        .exceptionally(thr -> handleException(thr))
        .thenAccept(data -> System.out.println(data));
}

private static Integer handleException(Throwable thr) {
    System.out.println("ERROR: " + thr);
    throw new RuntimeException("It is beyond all hope");
}

```

future.completeExceptionally(new RuntimeException()); - с две думи нищо не сме свършили

Състояния на completableFuture:

Pending (final)
Resolved state (final)
Rejected (final)

Future.orTimeout();

Combine and compose

//JavaScript
then(e => func(e)) ➔

In JavaScript func may return data or return a promise
If date is returned, it is wrapped into a promise
If promise is returned, then that is returned from the then

In JavaScript the return type could be any type, but in JAVA the return type should always be T (what we started with)

```

public static CompletableFuture<Integer> create(int number) {
    return CompletableFuture.supplyAsync(() -> number);
}
public static void main(String[] args) throws InterruptedException {
    create(2).thenCombine(create(3), (result1, result2) -> result1 + result2)
        .thenAccept(data -> System.out.println(data));
}

```

```

public static int[] func2(int number) {
    static {System.out.println("func2 initialized"); }
    return new int[] { number - 1, number + 1};
}

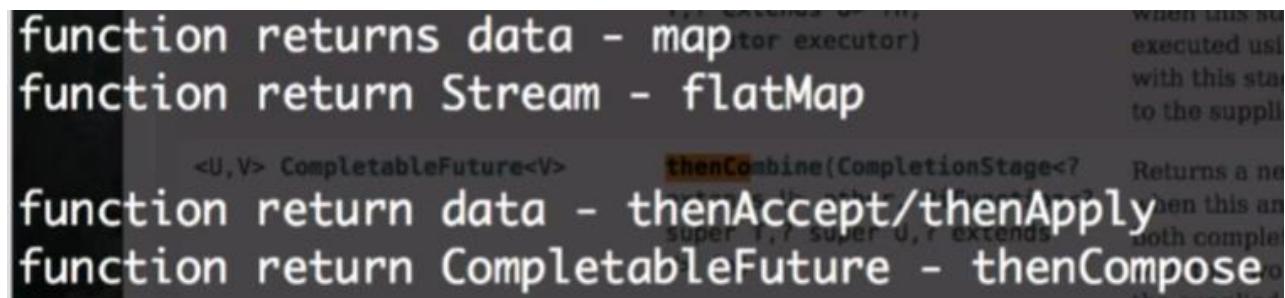
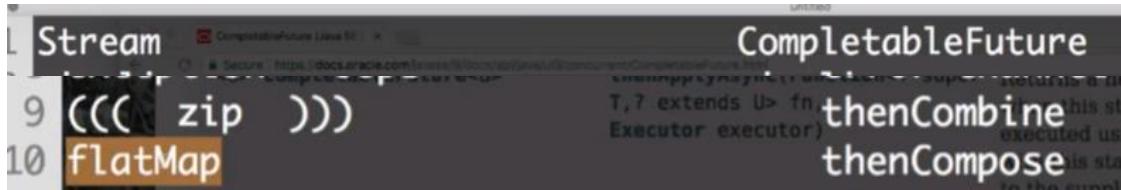
public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    numbers.stream()
        // .map(e -> func1(e))    //func1 is a one to one mapping function
        .map(e -> func2(e))      //func2 is a one to many mapping function
        .forEach(System.out::println);

    // map one-to-one Stream<T> ==> Stream<Y>
    // map one-to-many Stream<T> ==> Stream<List<Y>>
}

//flatMap one-to-many Stream<T> ==> Stream<Y> ???

```



```

public static CompletableFuture<Integer> create(int number) {
    return CompletableFuture.supplyAsync(() -> number);
}
public static CompletableFuture<Integer> inc(int number){
    return CompletableFuture.supplyAsync(() -> number + 1);
}

create(2)
    .thenApply(data -> inc(data))
    .thenCompose(data -> inc(data))
    .thenAccept(result -> System.out.println(result));

```

38. Text block from Java 13

Използва се с тройни кавички от двете страни.

```
private static final String json = """
{
    "country": "Bulgaria",
    "city": "Sofia",
    "street": "Mladost 4"
}""";
```

39. Object class

Когато е с == при обекти, то се сравняват по референция!!!

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

```
public class Object {
```

As far as is reasonably practical, the hashCode method defined by class Object returns distinct integers for distinct objects
{@IntrinsicCandidate}

```
public native int hashCode();
```

obj – the reference object with which to compare.

Returns: true if this object is the same as the obj argument; false otherwise.

API Note: It is generally necessary to override the hashCode method whenever this method **equals** is overridden, so as to maintain the **general contract for the hashCode method, which states that equal objects must have equal hash codes.**

```
public boolean equals(Object obj) {
    return (this == obj);
}
}
```

```
public final class System {
```

Returns the same hash code for the given object as would be returned by the default method hashCode(), whether or not the given object's class overrides hashCode(). The hash code for the null reference is zero.

Params: x – object for which the hashCode is to be calculated

Returns: the hashCode

Since: 1.1

See Also:

Object.hashCode, Objects.hashCode(Object)

```
@IntrinsicCandidate
```

```
public static native int identityHashCode(Object x);
}
```

```
public final class Objects {
```

```
public static int hashCode(Object o) {
    return o != null ? o.hashCode() : 0;
}
```

```
public static boolean equals(Object a, Object b) {
    return (a == b) || (a != null && a.equals(b));
}
}
```

Оператор == сравнява обекти по референция в паметта.

```
public class Main {
    public static void main(String[] args) { args: []
        String a = "AAA";
        String b = "BBB";
        MySuperObject a = new MySuperObject( sum: 25, name: "Svilen"); a: MySuperObject@700
        MySuperObject b = new MySuperObject( sum: 30, name: "Pesho"); b: MySuperObject@701
        var Aa :int = a.hashCode(); Aa: 1915318863
        var Bb :int = b.hashCode(); Bb: 1283928880
        var c = a == b = false; a: MySuperObject@700 b: MySuperObject@701
    }
}
```

Variables	
	p args = {String[0]@698} []
>	a = {MySuperObject@700}
>	b = {MySuperObject@701}
	f sum = 30
	f name = "Pesho"
o1	Aa = 1915318863
o1	Bb = 1283928880

провериха в паметта
hash ког

XX. други

```
stack.stream()
    .sorted(Comparator.reverseOrder())
    .map(obj -> String.valueOf(obj))
    .collect(Collectors.joining(", "));
```

native - ключова дума, изпълнява се на по-ниско ниво

Повтаря дадения стринг в случая 2 пъти

```
" ".repeat(2)
```

System.out.println(Collections.min(numbers)); - минималното от колекция (от тип stack)

Измерване на време – вариант 1

```
long start = System.currentTimeMillis();
run(нешо си);
long end = System.currentTimeMillis();
System.out.println(end - start);
```

Измерване на време – вариант 2

```
Date start = new Date();
doAlgorithm();
Date end = new Date();

System.out.println(end.getTime() - start.getTime());
```

12-04-1992

```
int[] partsOfDay = Arrays.stream(date.split("-")).mapToInt(Integer::parseInt).toArray();
LocalDate before = LocalDate.of(partsOfDay[2], partsOfDay[1], partsOfDay[0]);
```

или така

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
LocalDate before = LocalDate.parse(date, formatter);

DateTimeFormatter formatter = new DateTimeFormatterBuilder()
    .parseCaseInsensitive()
    .appendPattern("dd MMM yyyy")
    .toFormatter(Locale.US);
```

05 Dec 2021

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd MMM yyyy", Locale.US);
LocalDate after = LocalDate.parse(date, formatter);
```

В Java езика можем да пазим както елемента, така и предходния Node(връх)

```
public static class Node{
    private int element;
    private Node previous;
}
```

Алгоритъм за сортиране на данни Възходящо - ВАЖНО

```
for (int i = 0; i < list.getSize(); i++) {
    E current = list.get(i);
    for (int j = i+1; j < list.getSize(); j++) {
        E target = list.get(j);
        if (current.compareTo(target) > 0) {
            list.swap(i, j);
        }
    }
}
```

Пропускаме 1вият елемент

```
Arrays.stream(tokens)
    .skip(1)
    .mapToInt(Integer::parseInt)
```

```
.boxed()  
.toArray(Integer[]::new) – връща масив от бащин класа Integer[]
```

FlatMap – от матрица няколко реда от елементи го прави на общо 1 обект с всички елементи на матрицата

flatMap – прави от многомерни масиви на един поток/масив от елементи

Когато имаме Generic параметър, можем да подаваме примитивен **int** или **int[]** масив. Но изходния параметър е от бащин тип:

```
Function<int[], Integer> minFunction = .....
```

final String **key**; – когато използваме **final**, не можем да променяме референцията/стойността след това

Конвенция в JAVA за изписване на константи – само с главни букви

Конвенция в JAVA за записване на пакети packages – пишем с малка първа буква, за да избегнем дублиране с класове и интерфейси

```
name.trim().isEmpty(); = it cannot be null, empty or whitespace
```