

<https://docs.oracle.com/javase/tutorial/>

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+");
    }
}
```

- Чрез System.console()

```
String name = System.console().readLine(); - има екстра методи например къде да ми е курсора
```

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

"A good %s".formatted("idea") същото като `String.format("A good %s", "idea")`

`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 едноцифрено

```
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
4ри 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	9223372036854775807	64 bit

(-2^{63})

($2^{63}-1$)

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 стринг
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 закръгляния стават и чрез този пакет
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.ceil(23.45); // 24.0 - хвърля double като изчиства 24,00000000000000000003

Math.floor(45.67); // 45.0 - хвърля double като изчиства 45,00000000000000000003

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 каствем 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

Break and continue also have a labeled form that can be applied in case of nested loops

Чрез дефиниране на label **outer**, ще можем да излезем директно и от двата цикъла с **break outer;**

```

outer:
    for (int i = 1; i <= 100; i++) {
        for (int j = 1; j <= 100; j++) {
            x = x + i / 2;
            if (x > 1000) {
                break outer;
            }
            if (i % 17 == 0) {
                continue outer;
            }
        }
    }
}

```

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 b 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

Ctrl + Y - изтриване на маркиран/и ред/ове

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 – местя 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 й, т.н.

Ctrl + клик върху променлива, преди да сме въвели израза – показва информация за нея/типа й, 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 думи – част от файлове/класове

Ctrl+Shift+N : finds any file or directory by name

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

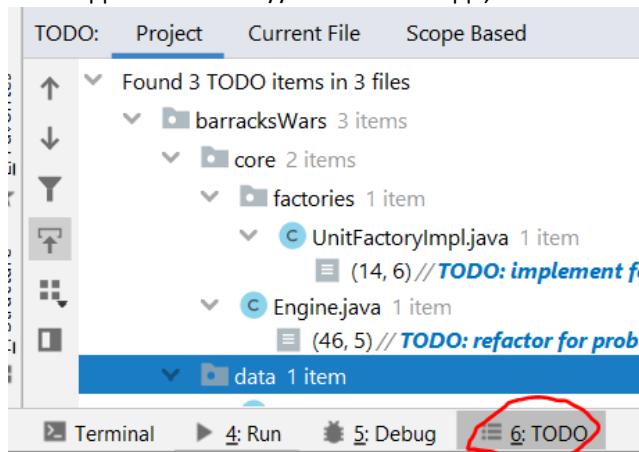
Snippets - Live Template – при маркиране на текст, за бърз достъп след това (от Tools):

Psvm – **public static void** main(String[] args)

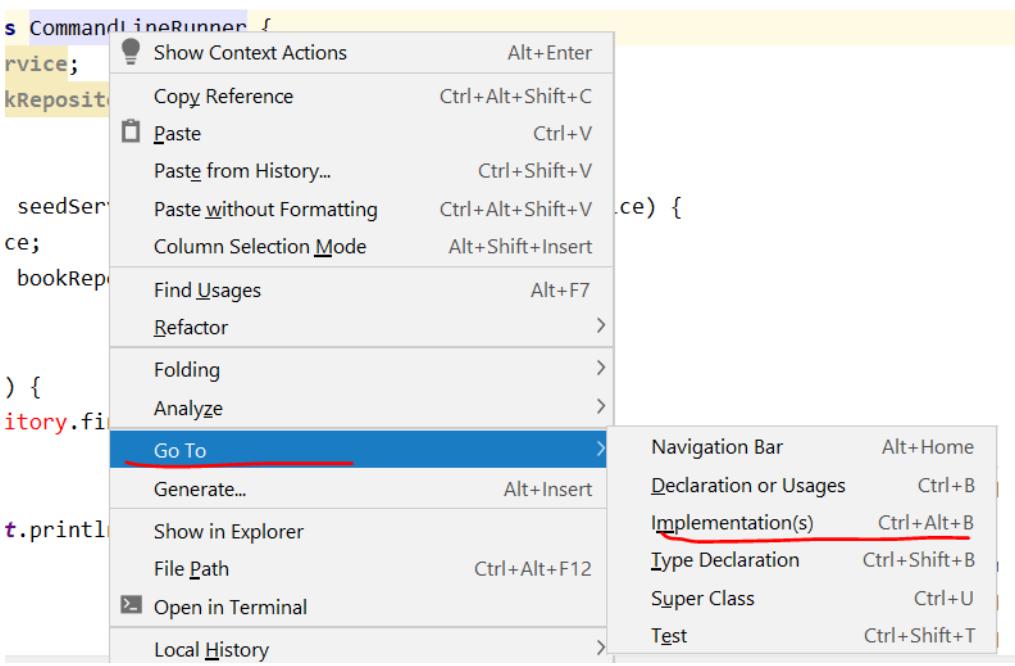
Sout – System.out.println("January");

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



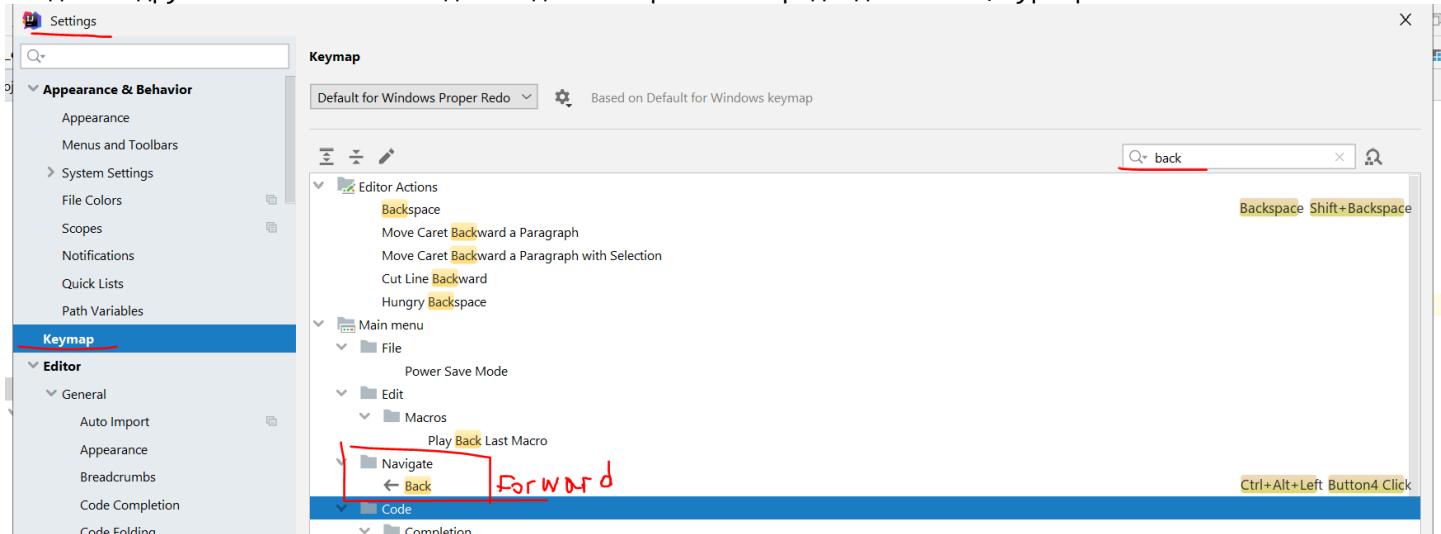
Go To Implementations



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);
```

4.2. Операции с масиви

Can we resize the array? - So, the answer is NO. But I answered YES, as we many times have "resized" arrays by creating new one with doubled its length.

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-a ☺
```

```
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 - спираме 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 /

Използва се за обхождане на масиви / листи / колекции - **read-only** е цикъла

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

foreach цикъла горе прави същото като нормален for цикъл – но служи само за обхождане и се води **read-only**.

```
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 сигнатурата!

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.



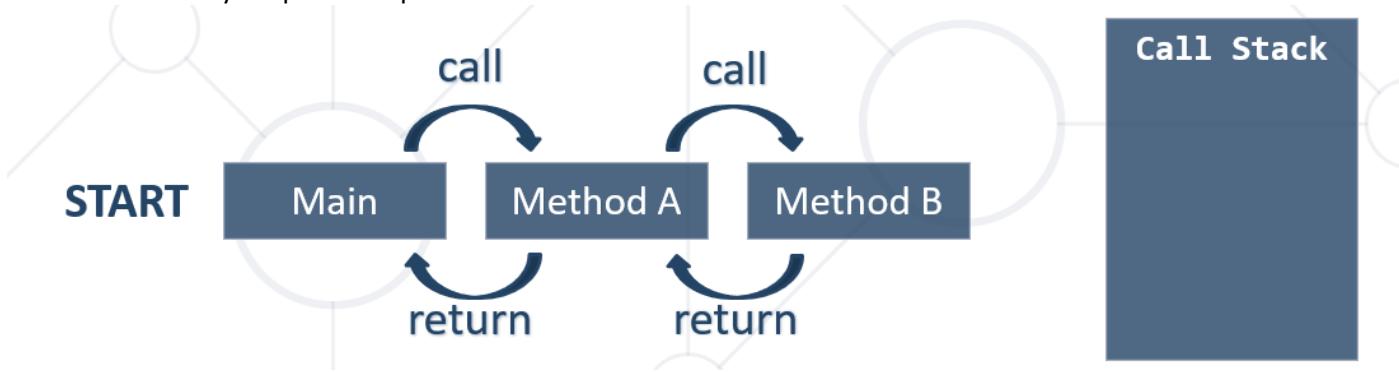
5.3. Debugging when Methods

Stack Frames показва на кое място (първо, второ) в момента на дебъгването/изпълнението на кода се изпълнява даден метод. Комбинацията от методи влизаат 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` е `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 элемент

```
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 списка в растящ ред да се преобразуват на 1 нов списък пак подреден в растящ 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 joinElementsByDelimiter(List<Integer> items, String delimiter) {
    String output = "";
    for (Integer element : items) {
        output += (element + delimiter);
    }
    return output;
}

String output = joinElementsByDelimiter(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 въче 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 – са public
- Constructors – it is a kind of Setter (Overloading default constructor; Constructor name is the same as the name of the class) – са 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 подгответни 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

When does a collision occur in a HashMap? - When two objects have the same result from hashCode() method!

Описание

Hash означава, че има уникални стойности на по-ниско ниво, и не се налага много 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.of - up to 10 key-value pairs

```

Map.of(pharmacist, createdPharmacist, operator, createdOperator, pharmacistAndOperator,
createdPharmacistAndOperator)
    .forEach(this::assertPersonnel);

```

Може да има конфликти при дублиране на елементи

```

Map<String, Personnel> persistedEntities = personnel.stream()
    .map(p -> transactionalService.runInTransaction(() -> persistPersonnel(p)))
    .collect(Collectors.toMap(k -> k.getEmail(), v -> v));

```

```

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 MAP

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()));
    }
}
```

Още методи с streams и map

```
-----  
Map<Long, UserBonus> userBonuses = profileBonuses.stream()  
.collect(Collectors.toMap(  
    ProfileBonusDTO::getBonusId,  
    UserBonus::new,  
    (a,b) -> a, LinkedHashMap::new)); //добави ги последователно
```

Returns a Collector that accumulates elements into a Map whose **keys** and **values** are the result of applying the provided mapping functions to the input elements.

```
Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,  
Function<? super T, ? extends U> valueMapper)
```

The map record is already created or if not we create one more record in the map, then we also return the value for that key.

```
wins.forEach(w ->  
    gameSummaryMap.computeIfAbsent(w.game(), game -> new PlayerGameSummaryDTO(game))  
        .setWinAmount(w.amount())  
);
```

Same as

```
wins.forEach(w -> {  
    if (gameSummaryMap.containsKey(w.game())) {  
        gameSummaryMap.get(w.game()).setWinAmount(w.amount());  
    } else {  
        gameSummaryMap.put(w.game(), new PlayerGameSummaryDTO(w.game(), w.amount()));  
    }  
})
```

Прави го на мап

```
final Map<Integer, List<BonusHistoryDTO>> grantedLoyaltyBonuses =  
Stream.of(bonusHistory)  
    .collect(Collectors.groupingBy(BonusHistoryDTO::fromLoyaltyLevel));
```

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

Definition and methods

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.

The Streams API

- The Streams API makes use of lambdas and extension methods
- Streams can be applied on collections, arrays, IO streams and generator functions
- Streams can be finite or infinite
- Streams can apply intermediate functions on the data that produce another stream (e.g. map, reduce)
- Streams are represented by a java.util.stream.Stream<T> instance
- Streams in the JDK can also be parallel

```
collection.stream();  
collection.parallelStream();
```

- Stream API methods include:
- filter(Predicate), map(Function), reduce(BinaryOperator), collect()
- sum(), min(), max(), count()
- anyMatch(), allMatch(), forEach()

Streams operations are composed into a pipeline

Streams are lazy: computation is performed when the terminal operation is invoked

```
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());
```

```
.peek in java.util.stream  
@BeforeEach  
@Transactional  
public void setUpEnvironment() {  
    pharmacyIds = Arrays.stream(Pharmacies.DATA)  
        .map(pharmacyMapper::toEntity)
```

```

    .peek(p -> p.persist())
    .map(p -> p.id)
    .collect(Collectors.toList());
}

```

Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. This is an intermediate operation.

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/презаписваме 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

```

mapLevelDTOs.entrySet().forEach(extraLevel -> new LoyaltyLevel(extraLevel.getKey(),
extraLevel.getValue()));
```

```

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((.....)
}

```

10.4. Stream API creating infinite streams

Java 8 Streams API - creating infinite streams with iterate and generate methods

<https://www.javabrahman.com/java-8/java-8-streams-api-creating-infinite-streams-with-iterate-and-generate-methods/>

Infinite Streams Streams are different from collections although they can be created from collections. Unlike collections, a stream can go on generating/producing values forever. Java 8 Streams API provides two static methods in the Stream interface for creating infinite streams. These are `Stream.iterate()` and `Stream.generate()`.

`Stream.iterate()`

Creating infinite Streams using the `Stream.iterate()` method:

Let us start by looking at the signature of `Stream.iterate()` method -

```
static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
```

Where,

- first input parameter is a seed value or initial value of type T
- second input parameter is a `UnaryOperator` function of type T
- output is a `Stream` of type T

`Stream.iterate()` method works just like a **function-of** algebraic operation which is commonly written as **f(x)**. The method first returns the seed-value itself. For the 2nd element in the Stream it finds **f(seed-value)** and from then on iteratively keeps applying **function-of** to the returned values.

So,

The 1st value in the infinite `Stream<T>` will be the **seed-value**

The 2nd value will be **f(seed-value)**.

The 3rd value will be **f(f(seed-value))**

The 4th value will be **f(f(f(seed-value))) and so on...**

Let us take an example to understand how `Stream.iterate()` method works -

Suppose the `UnaryOperator<T>` function `fsqr()` is a square function defined using the lambda expression - `(Integer n) -> n*n` and the `seed` is **2**.

So,

The 1st value returned in the infinite stream will be **2**

The 2nd value returned in the stream will be **fsqr(2) OR 2*2=4**.

The 3rd value will be **fsqr(fsqr(2)) i.e. fsqr(4) OR 4*4=16**

The 4th value will be **fsqr(fsqr(fsqr(2))) i.e. fsqr(16) OR 16*16=256 and so on...**

The infinite stream will then have values - **[2,4,16,256,... and so on...]**

Пример/задача:

```

public class Main {
    public static void main(String[] args) {
        java.util.stream.Stream<FibNumber> stream = fib();
        java.util.Iterator<FibNumber> iterator = stream.iterator();
        for (int i = 0; i < 4; i++) {
            iterator.next();
        }
    }
}

```

```

        System.out.println(iterator.next().getCurrent().intValue());
    }

static class FibNumber {
    private java.math.BigInteger previous;
    private java.math.BigInteger current;

    public FibNumber(java.math.BigInteger previous, java.math.BigInteger current) {
        this.previous = previous;
        this.current = current;
    }

    public java.math.BigInteger getPrevious() {
        return previous;
    }

    public java.math.BigInteger getCurrent() {
        return current;
    }
}

public static java.util.stream.Stream<FibNumber> fib() {
    return java.util.stream.Stream.iterate(
new FibNumber(new BigInteger("0" + "0"), new BigInteger(String.valueOf(1))),
    fibNum -> new FibNumber(
        fibNum.getCurrent(),
        fibNum.getPrevious().add(fibNum.getCurrent())))
);
}
}

```

[Stream.generate\(\)](#)

Creating infinite Streams using the `Stream.generate()` method

`Stream.generate()` method generates an infinite stream of elements by repeatedly invoking a [Supplier Functional Interface](#) instance passed to it as an input parameter. `Stream.generate()` method's signature looks like this -

static<T> Stream<T> generate(Supplier<T> s)

Where,

- Only input is an instance of a **Supplier Functional Interface** of Type T
- Output is a `Stream` of type T

```

public class InfiniteStreams {
    public static void main(String args[]) {
        Stream.generate(Math::random)
            .limit(5)
            .forEach(System.out::println);
    }
}

```

- `Math.random()` method generates a random value between `0.0` and `1.0` every time it is called. The [function descriptor](#) of `Math.random()` method matches that of the Supplier Functional Interface, so `Math.random()` is passed as an input to `Stream.generate()` method using its [method reference](#) - `Math::random`.

`Stream.generate()` method invokes `Math.random()` repeatedly to produce an infinite `Stream` of values between `0.0` and `1.0`.

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

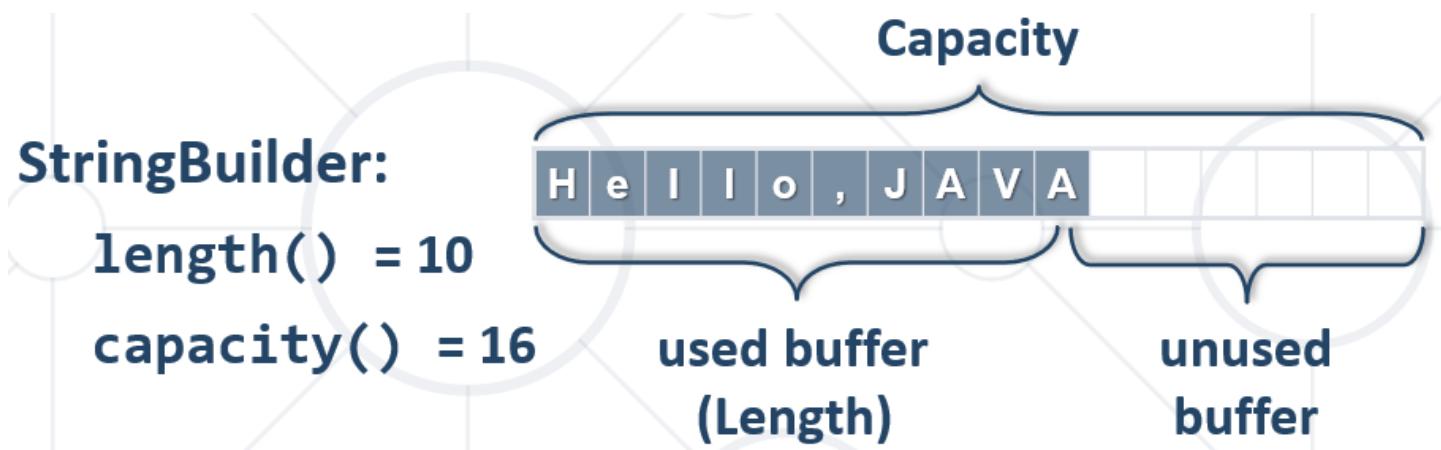
`@NotNull` – ако е constraint validation `@NotNull` включва проверка да не е и null.

`.isBlank()` – Ако е текстов стринг – има 2 библиотеки. Класическата `java.lang.String` метода `.isBlank()` не включва null проверка, но има библиотека от apache, която включва такава проверка

11.3. 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 Стинг 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());
```

11.4. Using the StringBuffer Class

A thread-safe, mutable sequence of characters - като StringBuilder, но thread-safe.
Ако по едно и също време се добавят няколко неща, и кое след кое да се добавя при такава ситуация.

```
StringBuffer sbuf = new StringBuffer();
sbuf.append("aaa").append("bbb");
```

11.5. 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://regexp.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 a b или c; втори символ d, e 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,} 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 ca 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 мачнати 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())
```

Pattern flags

The `Pattern.compile()` method may accept different flags (combined with | into a single integer) such as:

- `Pattern.CASE_INSENSITIVE`: ignores case while matching
- `Pattern.MULTILINE`: matches the entire text (and not line by line)
- `Pattern.ALL_FLAGS`: includes all flags
- Other flags in the `Pattern` class

```
int flags = Pattern.CASE_INSENSITIVE | Pattern.MULTILINE;
Pattern compile = Pattern.compile("^(.+)@(.+)\$", flags);
```

Methods of the `matcher` class include:

<code>matches()</code>	Checks if a string matches a pattern
<code>find()</code>	Finds the next occurrence of the match (returns false in case no more matches occur)
<code>start(n)</code>	Returns the start index of group n
<code>end(n)</code>	Returns the end index of group n
<code>pattern()</code>	Returns the Pattern for which this matcher applies
<code>group(n)</code>	Returns the group with index n
<code>replaceFirst()</code>	Replaces the first matched occurrence with a string
<code>replaceAll()</code>	Replaces all matches with a string

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+"; - отбелязва 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

Example

```

public class Main {
    public static void main(String[] args) {
        Interval interval = new Interval();
//        interval.init("1,2]");
        interval.init("[,2)");

        System.out.println(interval.isStartOpened());
        System.out.println(interval.getStart());
        System.out.println(interval.getEnd());
        System.out.println(interval.isEndOpened());
    }
}

public class Interval {
    private Boolean isStartOpened;
    private Boolean isEndOpened;
    private Integer start;
    private Integer end;

    public Interval() {
    }

    public void init(String interval) {
        String pattern = "^(\\[(\\d{0,10})\\d{0,10}\\])\\$";
        java.util.regex.Pattern regex = java.util.regex.Pattern.compile(pattern,
java.util.regex.Pattern.MULTILINE);
        java.util.regex.Matcher matcher = regex.matcher(interval);
        matcher.find();
    }
}

```

```

//           String group0 = matcher.group(0);
//           System.out.println(group0);

String group1 = matcher.group(1);
switch (group1) {
    case "" -> setStartOpened(null);
    case "(" -> setStartOpened(true);
    case "[" -> setStartOpened(false);
}

String group2 = matcher.group(2);
if (group2.equals("")) {
    setStart(null);
} else {
    setStart(Integer.parseInt(group2));
}

String group3 = matcher.group(3);
if (group3.equals("")) {
    setEnd(null);
} else {
    setEnd(Integer.parseInt(group3));
}

String group4 = matcher.group(4);
switch (group4) {
    case "" -> setEndOpened(null);
    case ")" -> setEndOpened(true);
    case "]" -> setEndOpened(false);
}
}
}

```

Stack and Queue

Elements in the stack follow the **First In Last Out (FILO) == Last In First Out (LIFO)** principle while in queue they follow the First In First Out (FIFO) principle.

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 [страница](#) 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--) {HE
    String child = children[i]; //HE
    stack.offer(child); //HE
}
for (String child : children) {
    stack.push(child);
}
```

// добавяме колекцията `tokens` в празната колекция `stack` – получава се обръната 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 дадения 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() + " ");
}

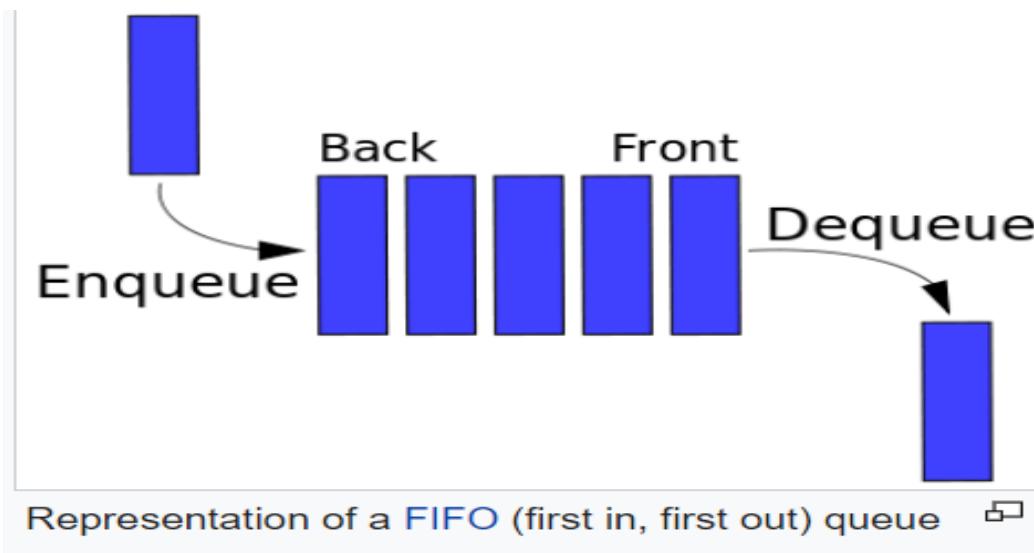
```

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)**. Новодошли 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.

Matrix R O W S	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]; - дължина на всяко измерение
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
е 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

Time complexity of a HashSet when contains operation - O(1) which is very clever implementation of a HashSet.

<https://stackoverflow.com/questions/6574916/hashset-look-up-complexity>

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 сортирана накрая, а да плащаме по-малко време с използване на Tree

LinkedHashSet<E> - The order of appearance is preserved

За TreeSet<E>

Червено-черно дърво – имплементира се в структурата TreeSet

1. Стари вариант до Java 7 – чрез мнимо имплементиране на интерфейса Comparator
Можем да използваме/връщаме и -7, +8 и той се оправя компаратора. Стига да не прехвърлим границата на типа числови данни.

Не е задължително да му връщаме само -1, 1 и нула, но по-добре точно 1, 1 и нула да му връщаме

```
Set<Person> people = new TreeSet<>(new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getAge() - o2.getAge();
    }
});
people.add(p1);
```

2. Новият вариант от Java 8 включително

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

3. Ако сравняваме обекти например от клас Person, можем да имплементираме в самия клас компаратора.

Слагането на компаратора в скоби на TreeSet конструктора се изпълнява с предимство спрямо имплементираното в класа Person compareTo метода

```

@Override
public int compareTo(Person o) {
    int ageResult = this.getAge() - o.getAge();
    if (ageResult != 0) {
        return ageResult;
    }

    return o.getName().compareTo(this.getName());
}

```

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 тази
    .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 тази
    .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);
}

s1.retainAll(s2);
So if we have two sets S1=[A, B, C] and S2=[A, B, E], the output of retain will be [A, B]. Given
that the set is a reference type. The retain function will remove all the entries which are not
contained in S1.

```

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() + ":");

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

```

Интелигентен начин да си вкараме първи елемент в даден мап

```

Map<String, Object> parameters = new HashMap<>() {{
    put("profileId", profileId);
}};

```

Когато имаме Мар с елемент Set като част от Мар-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 сътврение/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 MAP 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 от *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.

0. From String to InputStream

Which of the following classes can be used to convert String to an input stream? -

java.io.ByteArrayInputStream

```
String str = "Hello, world!";
InputStream is = new ByteArrayInputStream(str.getBytes("UTF-8"));
```

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 `newBufferedReader`

```

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);
```

В някои случаи може да се използва и следното:

Which of the following classes provides methods to write to a file? - java.util.Properties

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**

String path = "D:\\input.txt";

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

Wrapping a Stream

```

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 – c 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. Quick hint table

IO Classes

	Input	Output	Reader	Writer
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Formatted data		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		

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

5.0. Object serialization

Официално е вече deprecated 2023 – голям хак се е случил в Щатите

- Java SE provides a standard mechanism for implementing object serialization
- Classes of serializable objects need to implement the **java.io.Serializable** marker interface
- Objects can be serialized using a **java.io.ObjectOutputStream** instance
- Objects can be deserialized using a **java.io.ObjectInputStream** instance
- The object is serialized as a stream of bytes
- The JVM associates a serialization ID with each Serializable class for object compatibility
- If there is a field that is not serializable then a **NotSerializableException** is thrown

In many scenarios there is a problem we cannot modify the classes of some objects we refer(i.e. the ones coming from third-party libraries). In order to avoid that problem, we can define a custom serialization for the class by:

- Implementing the `readObject` and `writeObject` methods in the class

```
private void writeObject(ObjectOutputStream oos) {}
private void readObject(ObjectInputStream ois) {}
```

- Marking the non-serializable fields as **transient** (transient is also used by ORM frameworks such as Hibernate to mark fields as non-persistent)

```
public class Table implements Serializable {
```

```

private int size;
private transient Location location;

private void writeObject(ObjectOutputStream oos) throws IOException {
    //writes non-transient fields
    oos.defaultWriteObject();
}

private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {
    ois.defaultReadObject();
}

```

Standard Java(SE) serialization is not the only mechanism that can be used for object serialization...

Which of the following is **not true** about Java serialization?

- a.A class that is serializable must implement the java.io.Serializable interface
- b.Fields of a class that must not be serialized can be marked as transient
- c.If a class is not serializable an unchecked exception is thrown**
- d.During serialization the object must be converted to a stream of data

5.1. *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);
}
```

5.2. 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);
}
```

5.3. 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();
}
```

7. 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 byteContainer;
    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();
}

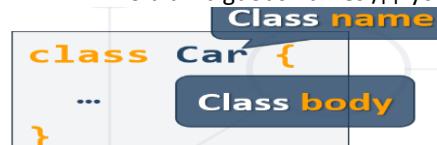
```

18. Defining Classes

1. Defining Classes

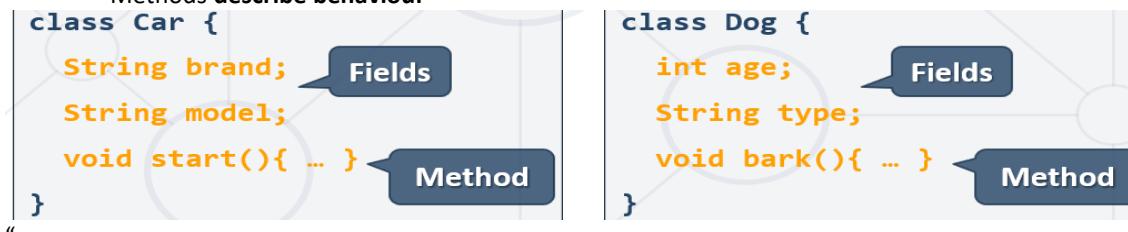
Naming Classes

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

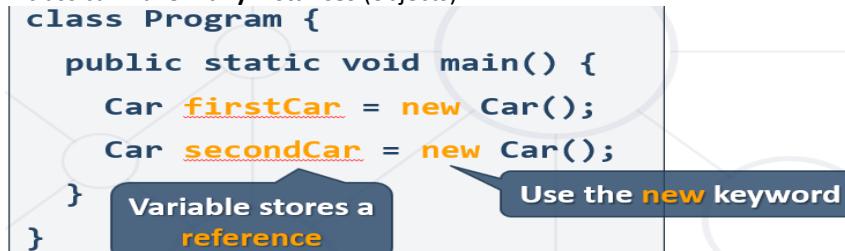


Class is made up of **state** and **behavior**:

- 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

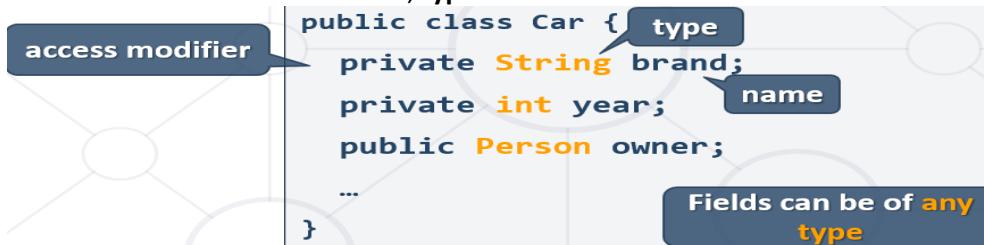
Classes provide structure for creating objects

An object is a single instance of a class

2. Class Data

Fields

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



Access 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 {
    private int horsePower;
    public int getHorsePower() {
        return this.horsePower;
    }
    public void setHorsePower(int horsePower) {
        this.horsePower = horsePower;
    }
}

```

Field is hidden

Getter provides access to field

this points to the current instance

Setter provide field change

Keyword **this**

- Prevent field hiding
- Refers to a current object

Ако не използваме **this** в сетъра `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 е 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 презписва 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);
        }
    }
}

Class MyApp {
    // 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();
    }
}
```

Инстанциране на вътрешни класове

Нормално описание на един клас в друг клас - например package-private

```
public static void main(String[] args) {
    Principle1 principle1 = new Principle1();
    Principle1.Car car = principle1.new Car();
    car.engine = new Object();

    Principle1.CarRepairSystem carRepairSystem = principle1.new CarRepairSystem();
    carRepairSystem.isEngineGood(car);

    carRepairSystem.printReceipt();
}
```

8. Records since Java 15

Record classes are special classes that act as transparent carriers for immutable data. They are immutable classes (all fields are **final**) and are implicitly final classes, which means **they can't be extended!**

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){}
```

9. Trying to mix it results in error

What is the result of the following? - compilation error

```
A a = new A();
A b = new B();
B c = (B) a;
```

ClassCastException

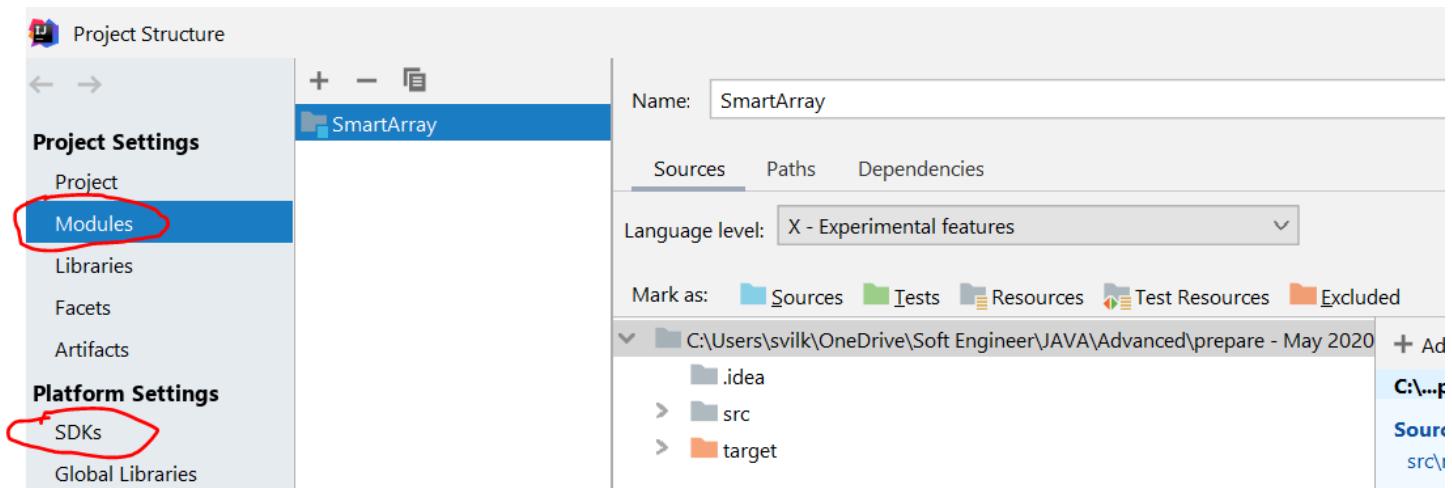
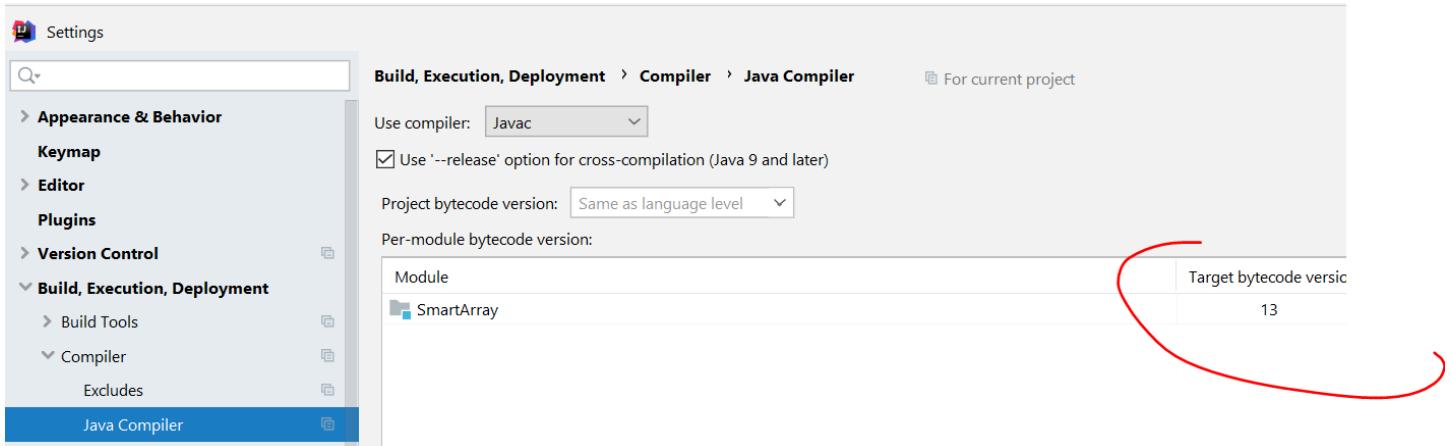
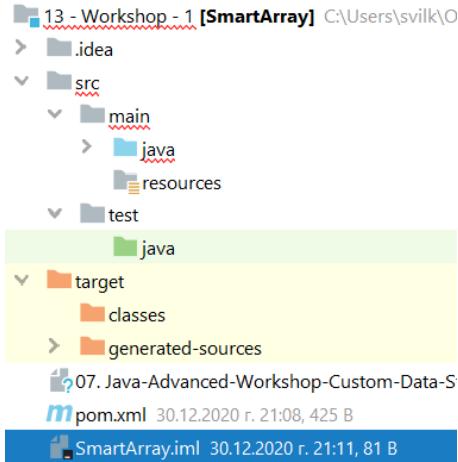
Main.java:16:15 java: incompatible types: B cannot be converted to A

Main.java:17:19 java: incompatible types: B cannot be converted to A

19. Статична или динамична реализация – разработване на клас + 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`)

int[]	(n)	Mode	Cnt	Score	Error	Units
testAddInArrayList	10000000	avgt	5	6.802 ± 5.514	ms/op	
testAddInStack	10000000	avgt	5	12.284 ± 19.422	ms/op	

| with exit code 0 new Node

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 четееме записани елементи
this.elements[this.size() - 1] = 0; //this.index винаги е с един повече
this.index--; //намаляме мястото, до което четееме 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.

Which of the following classes is not an implementation of the **List** interface?

- a. Vector
- b. ArrayList
- c. **ArrayDeque**
- d. LinkedList

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**.

При HashSet и HashMap

2. Различните структури от данни, освен че запаметяват много данни последователно или произволно те имат и различна сложност на операциите за търсене, добавяне и изтриване

- може да афектира бързодействието (производителност) на приложението
- ако не бъдат използвани правилните структури за правилните операции
- трябва да си нахвърлим някаква идея какви и колко често операции ще се случват

3. При списъците – добавяне в началото и средата, ако се случва често – не е страшно добра идея да ползваме ArrayList.

4. Множества и най-вече хеширащо множество (HashSet)

- Всички множества имат уникални елементи
- Множеството като абстракта структура от данни (ADT) не дефинира какъв е редът на появя
- Множествата имат различни имплементации
 - * HashSet -> който използва hashCode за да сложи елемент в някоя позиция от масива съответно и по този начин да го намери, ако питаме дали го има (contains)
 - > може да има колизии – два елемента да дадат един и същ хеш код или техния дори и различен хеш код да попадне на еднакво място в масива (hashCode % 16 == x)
 - > Трябва да разрешим колизията, хеш сетът го прави от самосебе си (java.util.HashSet) и използва алгоритъм в който или да ги редистрибутира или да използва свързан списък по който да търси с equals
 - > Трябва да сме имплементирали правилно едновременно hashCode и equals

Ако модулното делене на примерно 16 за различни хешкодове даде един и същи резултат/остатък, тогава едни и същи хешкодове се групират на bucket-и и влиза в действие **equals** функцията.

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 каstne експлицитно към даден тип (примитивни) данни.

Generics - оказваме параметъра T на нестатичен метод както при параметрите, така и като тип на метода (отпред пред void):

```
private <T> void publishMessage(String identifier, String type, T payload, String extraComment) {
    String payloadString = JsonUtils.toJsonString(payload);
    var kafkaMessage = KafkaRecord.of(identifier, payloadString).withHeader("type", type);
    logger.infof("Publishing message of type %s for userId %s to FastTrack topic - %s",
    type, identifier, extraComment);
    fastTrackEmitter.send(kafkaMessage);
}
```

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 използвамения 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>,
    a 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 тънда 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 Inference

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the most specific type that works with all of the arguments.

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();  
can be substituted with  
Map<String, List<String>> myMap = new HashMap<>(); you must use the diamond
```

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

can be substituted with

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

20.6. Type Erasure – типово изтриване

Info

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 компиляция 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 компиляция, 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 – Т не съществува след компиляцията

```
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
    }
}
```

Erasure of generic types

During the type erasure process, the Java compiler:

- erases all type parameters
- and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

```
public class Node {
```

```

private Object data;
private Node next;

public Node(Object data, Node next) {
    this.data = data;
    this.next = next;
}

public Object getData() { return data; }
// ...
}
-----
```

In the following example, the generic `Node` class uses a bounded type parameter:

```

public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

The Java compiler replaces the bounded type parameter `T` with the first bound class, `Comparable`:

```

public class Node {

    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // ...
}
```

Erasures of generic methods

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

```

// Counts the number of occurrences of elem in anArray.
// 
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
    return cnt;
}
```

Because `T` is unbounded, the Java compiler replaces it with `Object`:

```

public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray)
```

```

    if (e.equals(elem))
        ++cnt;
    return cnt;
}
-----

```

Suppose the following classes are defined:

```

class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }

```

You can write a generic method to draw different shapes:

```
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

The Java compiler replaces T with Shape:

```
public static void draw(Shape shape) { /* ... */ }
```

Bridge Methods

<https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html>

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, which is called a bridge method, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

20.7. Generics, inheritance and subtypes

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn!

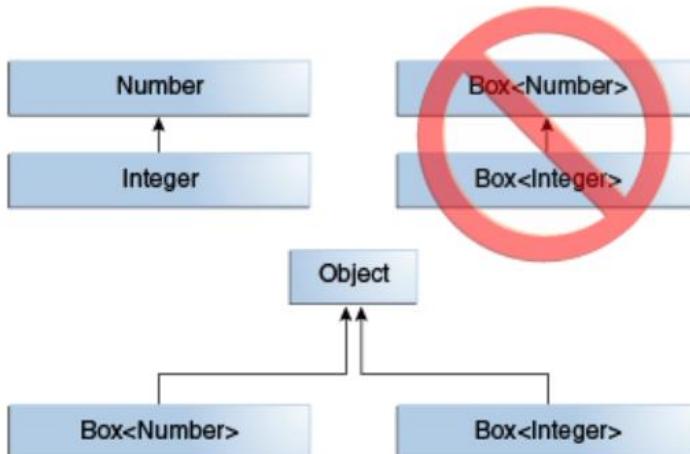
```

public class Box<T> {
    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}

```



`Box<Integer>` is not a **subtype**(**child class == subclass**) of `Box<Number>` even though `Integer` is a subtype of `Number`.

20.8. 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!

Generics с използване на `extends` (wildcard ? с използване на `extends` или `super`)
Правила:

- можем да екстендирам **само 1 клас**, но можем да унаследим много интерфейси.
- за този конкретен синтаксис винаги използваме `extends` (или `super`) като първо задължително се цитира (абстрактния) клас, и чак след това интерфейсите. Разделят се със знака &

```
//Multi bound,  
no usages  ▲ Aleksandar Vladimirov  
static class Cheetach<D extends GrassEaters & Animal & Cloneable & Serializable> {  
  
}  
  
no usages  new*  
static class Lion<D extends BaseAnimal & GrassEaters & Animal> {  
    ...  
}
```

```
Class A { /* ... */ }  
interface B { /* ... */ }  
interface C { /* ... */ }  
  
class D <T extends A & B & C> { /* ... */ }
```

If bound A is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ } // compile-time error
```

`<T extends Animal>`

Клас Cat екстендира класа Animal

`<T extends Class>` - specifies an "Upper/parent bound" - `class Cat<T extends Animal>`

В този случай, Т 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 екстендира 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;

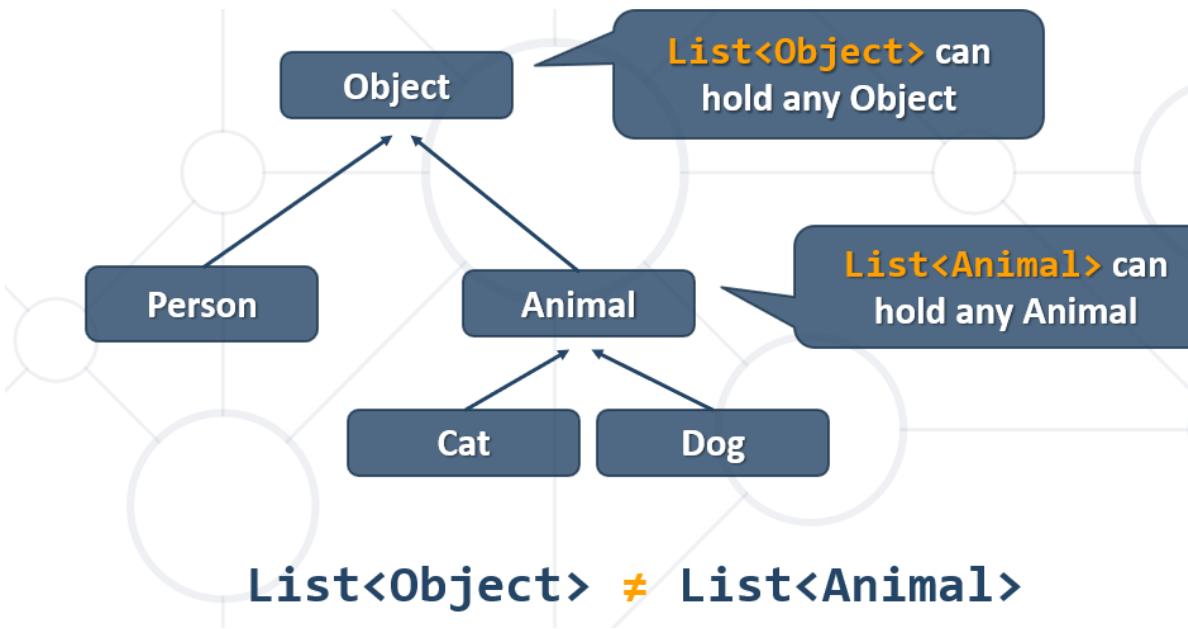
    @Override
    public int compareTo(E o) {
        return data.compareTo(o);
    }
}
```

20.9. 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!!!



20.10. 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;
    }
}

```

20.11. Примерна задача

Write a generic method `findMax` that returns the largest element in a generic array of elements (vararg method parameter) of a boxing type (corresponding to one of the 8 primitive types) and String. For strings the largest element is the lexicographically largest element.

```

public class Main {
    public static void main(String[] args) {
        System.out.println(findMax("1", "2", "3"));
        System.out.println(findMax(1, 2, 3));
        System.out.println(findMax("abx", "z", "bz"));
        System.out.println(findMax(false, true, true, false));
        System.out.println(findMax('a', 'c', 'd', 'a'));
        System.out.println(findMax(2.0, 3.6, 2.4, 2.6));
    }

    @SuppressWarnings("unchecked")
    public static <R extends Object> Object findMax(R... params) {
        R maxResult = params[0];

```

```

        for (int i = 0; i < params.length - 1; i++) {
            if (maxResult instanceof Integer) {
                if ((Integer) maxResult < (Integer) params[i + 1]) {
                    maxResult = params[i + 1];
                }
            } else if (maxResult instanceof Boolean) {
                if ((Boolean) params[i + 1]) { //==true
                    maxResult = params[i + 1];
                }
            } else if (maxResult instanceof String) {
                if (((String) maxResult).compareTo((String) params[i + 1]) < 0) {
                    maxResult = params[i + 1];
                }
            } else if (maxResult instanceof Character) {
                if (((Character) maxResult).compareTo((Character) params[i + 1]) < 0) {
                    maxResult = params[i + 1];
                }
            } else if (maxResult instanceof Double) {
                if (Double.compare((Double) maxResult, (Double) params[i + 1]) < 0) {
                    maxResult = params[i + 1];
                }
            }
        }

        return maxResult;
    }
}

```

20.12. Вместо Generics

Вместо да използваме Generics или друга форма на организация на класовете, в случаите когато искаме да изпратим дадени данни към външен сервис, то:

1. Можем да използваме Map<String, Object>

```

public class CustomMessage{
    public @JsonProperty("notification_type") String notificationType;
    public @JsonProperty("user_id") String userId;
    public String origin;
    public String timestamp;
    public Map<String, Object> data = new HashMap<>();
}

```

2. Важно е да отбележим че намапването към крайния обект (в случая CustomMessage) трябва да стане след като са се изпълнили бизнес логика проверки. В противен случай рискуваме да правим каствания, което не е ок.

```

publishMessage(dataDTO.userId, type: "CUSTOM", dataDTO, (boolean) dataDTO.data.get("isImpersonated") ? "player_online" : "player_online from user logout");

```

21. Wild cards and Generics

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 / a list of any 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);
    }
}

```

Example bound limitations with generics

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)!
Само при wildcard можем да използваме и `super` като допълнителна опция.

Is the following definition valid: true

```
public <R extends Runnable> Object execute(R task) { .. }
```

Is the following definition valid: false

```
public <R super Runnable> Object execute(R task) { .. } – няма супер, няма родител на Runnable
```

Guidelines for Wildcard Use

An "In" Variable

An "in" variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the "in" parameter.

An "Out" Variable

An "out" variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the "out" parameter.

Of course, some variables are used both for "in" and "out" purposes.

Wildcard Guidelines:

- An "in" variable is defined with an upper bounded wildcard, using the `extends` keyword.
- An "out" variable is defined with a lower bounded wildcard, using the `super` keyword.
- In the case where the "in" variable can be accessed using methods defined in the `Object` class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

22. Iterators and Comparators

22.1. 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; //массив
```

```

        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 някакви 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 коструктор с елементи

```

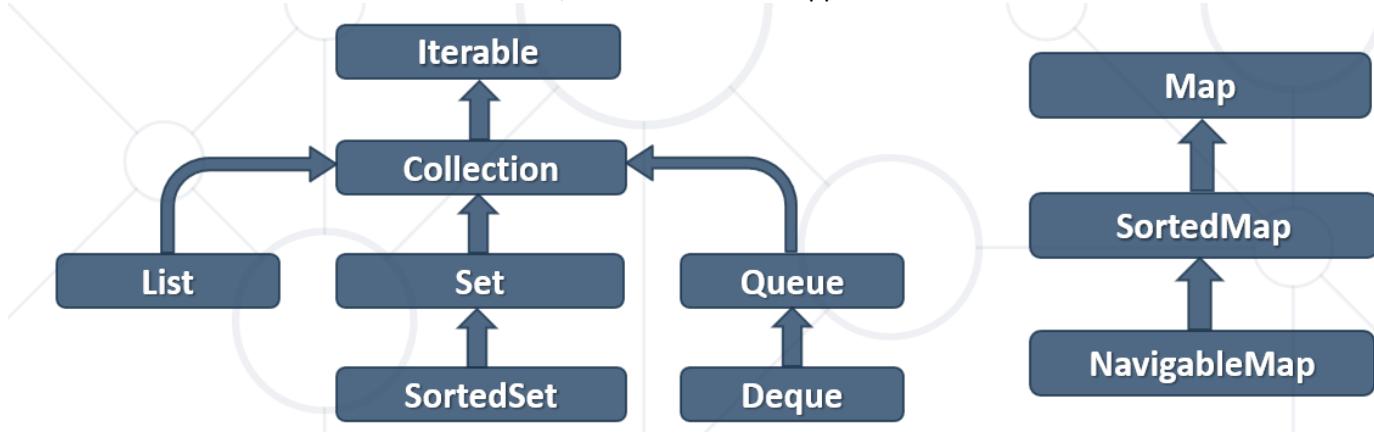
public class MessageLogger implements Logger {
    private Set<Appender> appenders;

    public MessageLogger(Appender... appenders) {
        this.setAppendlers(appenders);
    }
...
public class Main {
    public static void main(String[] args) {
        Logger logger = new MessageLogger(); //нулев конструктор позволява!!!

```

22.2. Iterable<T> and Iterator<T> interfaces

Inheritance leads to hierarchies of classes and/or interfaces in an application:



1) *Iterable<T>* - каза 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.setAuthors(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"
    └─ f id = 2
    > └─ f username = "pesho_new_new"
    └─ f age = 25
    > └─ f registrationDate = {LocalDate@2636} "2022-02-22"
    > └─ f lastLoggedIn = {LocalDate@2637} "2022-02-22"
  └─ 1 = {User@2632} "User{id=3, username='Svilen', age=36,
    └─ f id = 3
    > └─ f username = "Svilen"
    └─ f age = 36
    > └─ f registrationDate = {LocalDate@2641} "2022-02-22"
    > └─ f 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>

What is an iterator? - An object used to loop through collections.

```

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<>
    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<>
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<E>
public Iterator<E> iterator() {
    return this.books.iterator();
}

```

Разглеждане на обикновен Iterator<E> 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<E> (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);
    }
}

```

22.3. Comparable<T> and Comparator<T>

1. Comparable<T>

Казва, че елементите на класа имплементиращ Comparable<T> 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 Comparable<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);
    }
}

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<E>

Multiple sorting sequence

Doesn't affect the original class

What is a comparator ?

- a.all of the above
- b.An object that can be used to provide sorting mechanism for Java collections
- c.Comparison functions provided by an implementation of the java.util.Comparator interface
- d.An object used to provide a comparison function between two other objects

compare() method - сравнение левия с десния – подобен метод на compareTo – създаваме си собствена логика на сравнение за дадена структура от данни

Вариант 1: - ламбда мнима функция и мним клас на Comparator<E>

```

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());
}
или с Comparator.comparing и thenComparing
}
}

```

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 дефолтна сортиранка, и тя е валидна за който и да е chaining на stream-а.

Коригираме метода на Iterable<Book>, тогава сортиранката 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 сортиранка!!!

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)) { //обект object 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());
}
```

23. Functional programming - expressions

23.1. Functional interface - definition

Java Functional Interfaces

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, [lambda expressions](#) can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. **Runnable**, **ActionListener**, and **Comparable** are some of the examples of functional interfaces.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called @FunctionalInterface**. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only. We can also call Lambda expressions as the instance of functional interface.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

```
class Test {
    public static void main(String args[])
    {
        // create anonymous inner class object
        new Thread(new Runnable() {
            @Override public void run()
            {
                System.out.println("New thread created");
            }
        }).start();
    }
}
```

Java 8 onwards, we can assign [lambda expression](#) to its functional interface object like this:

```
// Java program to demonstrate Implementation of
// functional interface using lambda expressions
class Test {
    public static void main(String args[])
    {

        // lambda expression to create the object
        new Thread(() -> {
            System.out.println("New thread created");
        }).start();
    }
}
```

`@FunctionalInterface` annotation is used to ensure that the functional interface can't have more than one abstract method. In case more than one abstract methods are present, the compiler flags an 'Unexpected `@FunctionalInterface` annotation' message. **However, it is not mandatory to use this annotation.**

Below is the implementation of the above topic:

```
// Java program to demonstrate lambda expressions to
// implement a user defined functional interface.

@FunctionalInterface
interface Square {
    int calculate(int x);
}

class Test {
    public static void main(String args[])
    {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x) -> x * x;

        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interfaces. All these interfaces are annotated with `@FunctionalInterface`. These interfaces are as follows:

- **Runnable** → This interface only contains the `run()` method.
- **Comparable** → This interface only contains the `compareTo()` method.
- **ActionListener** → This interface only contains the `actionPerformed()` method.
- **Callable** → This interface only contains the `call()` method.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

1. **Consumer**
2. **Predicate**
3. **Function**
4. **Supplier**

Amidst the previous four interfaces, the first three interfaces,i.e., Consumer, Predicate, and Function, likewise have additions that are provided beneath –

1. Consumer -> Bi-Consumer
2. Predicate -> Bi-Predicate
3. Function -> Bi-Function, Unary Operator, Binary Operator

Which of the following is not a functional interface?

- a.java.lang.Cloneable
- b.java.lang.Runnable
- c.java.util.function.UnaryOperator
- d.java.lang.Comparable

23.2. 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 тип, 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));

```

23.3. 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> parseInt = x -> Integer.parseInt(x);
int[] numbers = Arrays.stream(sc.nextLine().split(", "))
    .mapToInt(parseInt)
    .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() //16
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));

```

CurriFunction

```

public class CurriFunctionDemo {
    public int multiply(int a, int b){
        return a * b;
    }

    Function<Integer, Function<Integer, Integer>> currMulti = u -> v -> u * v;

    public void test(){
        int res1 = multiply(2, 3);
        int res2 = currMulti.apply(2).apply(3);
    }
}

```

Higher-order function

```

//Higher-order function . takes one or more functions as arguments
public class HighOrder {
    public void exampleTakesParam(){
        List<String> students = new ArrayList<>();
        students.add("Student1");
        students.add("Student2");
        students.add("Student3");

        //sort е high-order функция
        students.sort((a, b) -> a.compareTo(b));

        //функцията comp връща друга функция comp.reversed, която пак е Comparator
        Comparator<String> comp = (a, b) -> a.compareTo(b);
        comp.reversed();
    }
}

```

Pure functions

```

public class Purity {
    //pure function - при един и същи input връщат един и същи output - referential transparency
    // An expression is called referentially transparent if it can be replaced with its
    corresponding value without changing the program's behavior.
    public int sum(int a, int b) {
        return a + b;
    }

    int requestCount = 0;

    //non-pure function - Модифицира някакъв state
    public void incRequest(int count) {

```

```

    requestCount += count;
}

//constant
public int get(){
    return 10;
}

//не връща резултат, и е безполезна
public void useless(int a) {
    int b = a + 1;
}
}
}

```

23.4. 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);
}

```

Several Consumers actions

This is the lazy way to not edit all methods calling this method without doing overloading

```
createWallets(List.of(new CreateWalletDTO(profileId, "EUR"), new CreateWalletDTO(profileId, "AED"))),
    wallet -> {
        wallet.lifetimeWithdrawal = new BigDecimal("300.00");
        wallet.lifetimeDeposit = new BigDecimal("200.00");
        wallet.balance = new BigDecimal("100.00");
    });
}

List<Long> createWallets(List<CreateWalletDTO> wallets, Consumer<Wallet>... additionalWalletConfig)
{
    List<Long> walletIds = new ArrayList<>();

    for (CreateWalletDTO walletDTO : wallets) {
        Wallet wallet = new Wallet();
        wallet.currency = walletDTO.currency();
        wallet.profileId = walletDTO.profileId();

        for (var config : additionalWalletConfig) {
            config.accept(wallet);
        }
    }
}
```

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();
}

```

23.5. 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);
```

Another example BiFunction

```
public class Main {
    static int sum(int x, int y) throws IOException {
        // validations on x and y ...
        return x+y;
    }

    public static void main(String[] args) throws IOException {
        BiFunction<Integer, Integer, Integer> f = (t, u) -> {
            try {
                return sum(t, u);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        };
        System.out.println(f.apply(10, 20));
    }
}
```

Вариант нормална функция с входен параметър масив

```
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());
```

Друг пример за нормална функция, където входен и изходен параметър са `List<Integer>` и имаме в допълнение и `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);
    }
}

```

23.6. 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.getValue()));

String format = sc.nextLine();

people.stream()
    .filter(predicateMap.get(ageCondition))
    .forEach(consumerMap.get(format));

```

23.7. 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.getKey())))
                printer.accept(person);
        }
    }
}

```

23.8. 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

23.9. 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

Basic principles of object oriented programming - Abstraction, encapsulation, polymorphism, inheritance
PIE (Polymorphism, Inheritance, Encapsulation)

Can you override private or static methods in Java? - none of them can be overridden

24. Working with Abstraction

Абстракция не може да се дефинира с конкретика. Точно затова конкретизираме създавайки повече класове и обекти.

24.1. 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

24.2. 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

24.3. 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**

Enum - ако добавим ново поле за enum, то нарушаваме open-close принципа. Например ако сортираме по ординал може да се объркат сметките.

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
Като имаме вече даденото поле на Season и на Discount, то можем да викаме
season.ordinal() - държавен
season.name() = season.toString() - държавен
season.getSeasonIndex() - дефиниран getter в enum класа Season

        System.out.printf("%.2f", getPriceFor(pricePerDay, numberOfDays, season, discount));
    }

    private static double getPriceFor(double pricePerDay, int numberOfDays, Season season,
Discount discount) {
        return pricePerDay * season.getSeasonIndex() * numberOfDays * discount.getDiscount();
    }
}

```

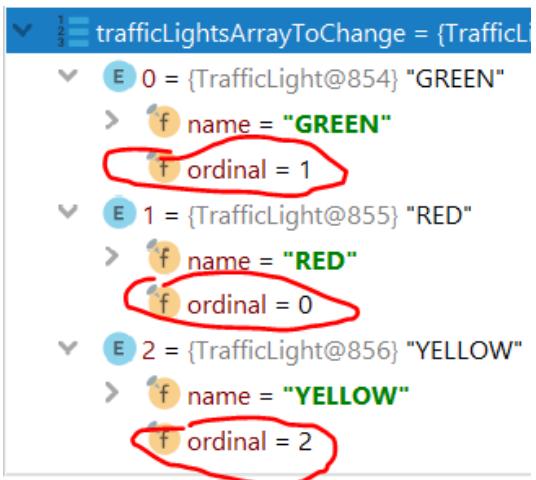
Друг пример

```

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

```



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")
}

```

```

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

24.4. 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

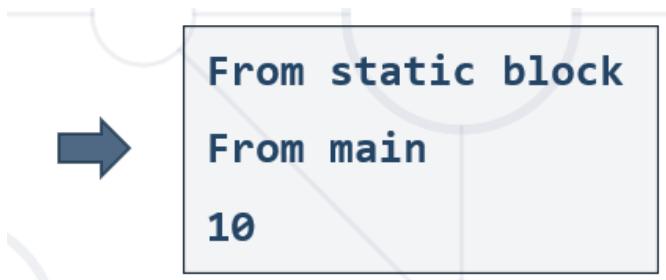
- Belongs to the class rather than the object of a class
- Can be **invoked** without the need for creating an instance of a class
- Can **access** static data member and can **change** the value of it
- Can **not use non-static** data member or call **non-static method** directly
- **this** and **super** 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");  
    }  
}
```

Static block 1
Static block 2
Main block

И още един по-интересен пример!:

What is the result of calling B b = new B() ?

```
public class A {  
    public A() {  
        System.out.println("1");  
    }  
}
```

```

        }
    }

public class B extends A {
    static {
        System.out.println("2");
    }
    {
        System.out.println("3");
    }
    public B() {
        System.out.println("4");
    }
}

```

2 first execute the static block

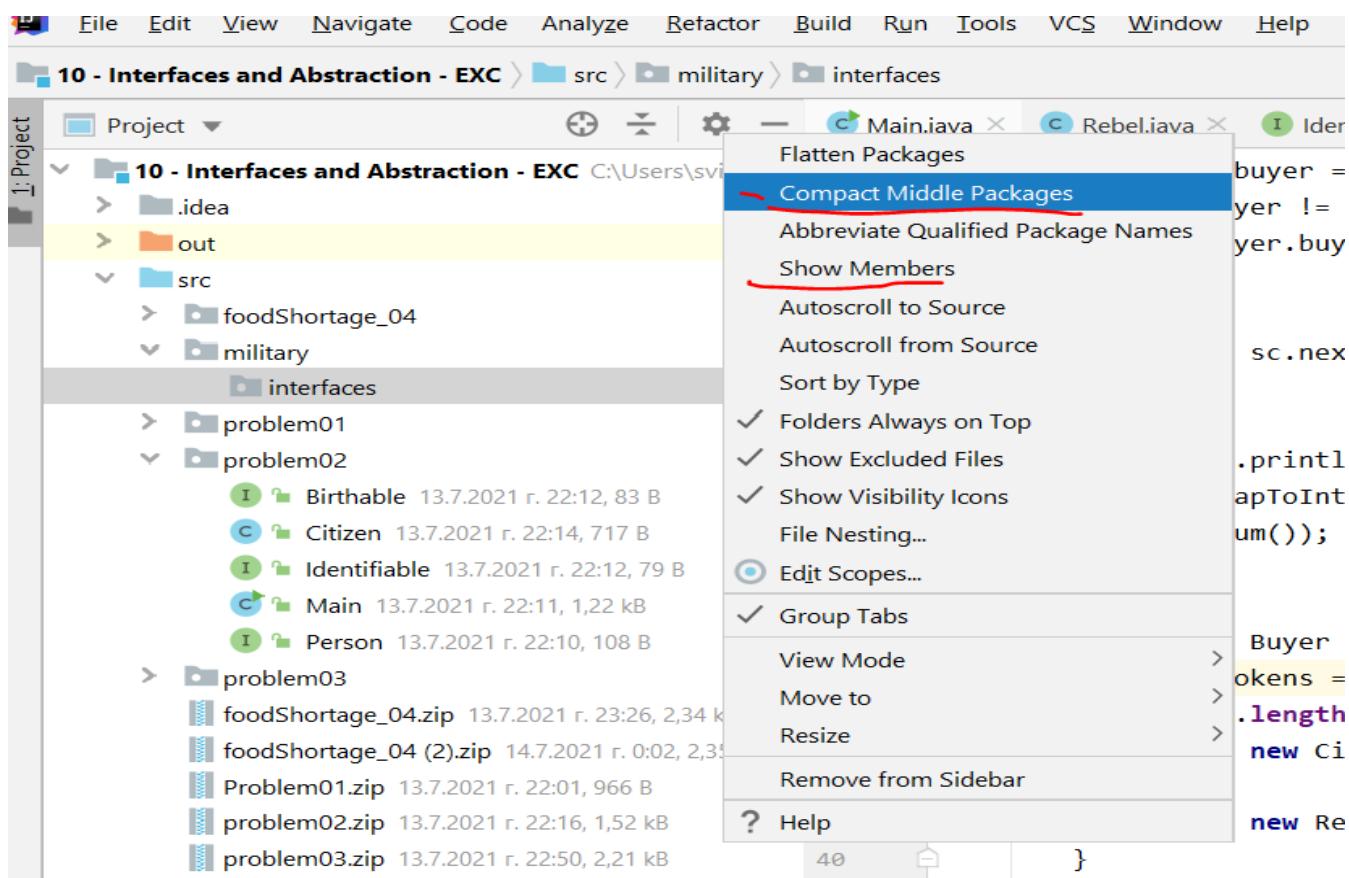
1 then execute the constructor of the parent class

3 then execute the normal block

4 finally execute the constructor of B class

24.5. 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

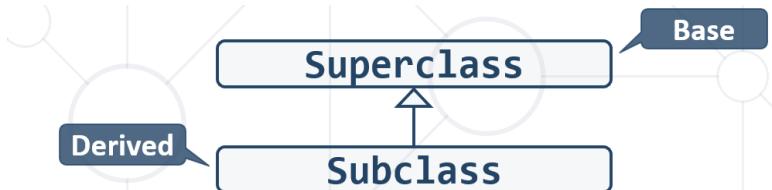
- The library is divided into packages and classes
- Import a single class or a whole package that contain all the classes
- To use a class or a package, use the import keyword
- The complete list can be found at Oracles website: <https://docs.oracle.com/en/java/javase/>

```
import package.name.Class; // Import a single class
import package.name.*;    // Import the whole package
```

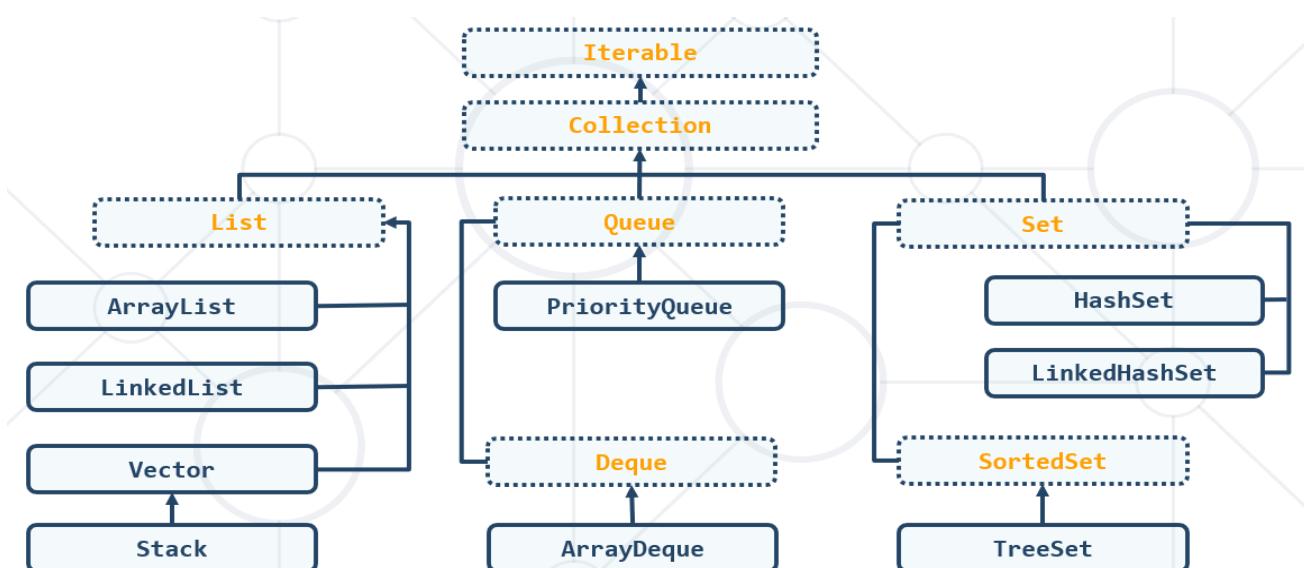
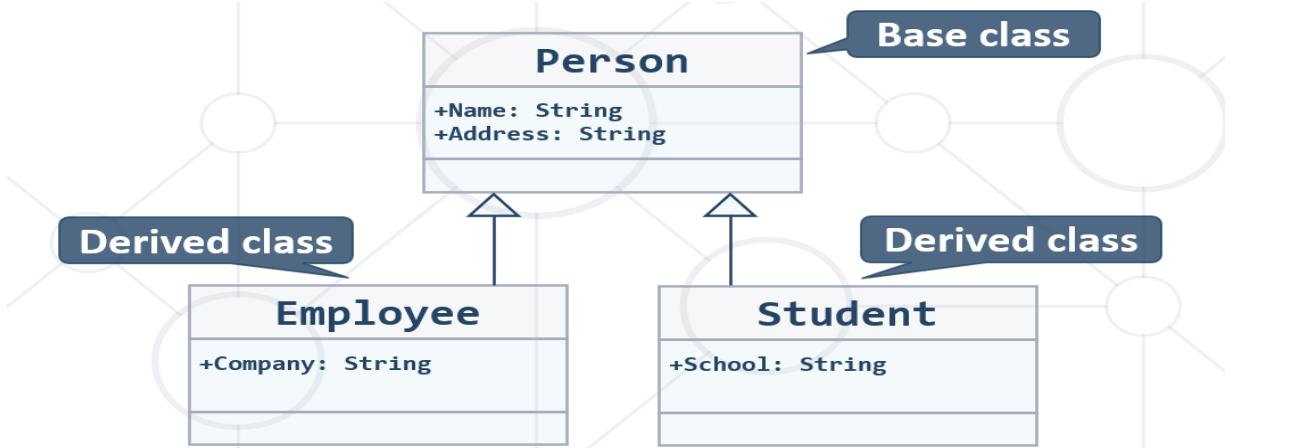
25. Inheritance – extending classes

25.1. 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 { ... }
```

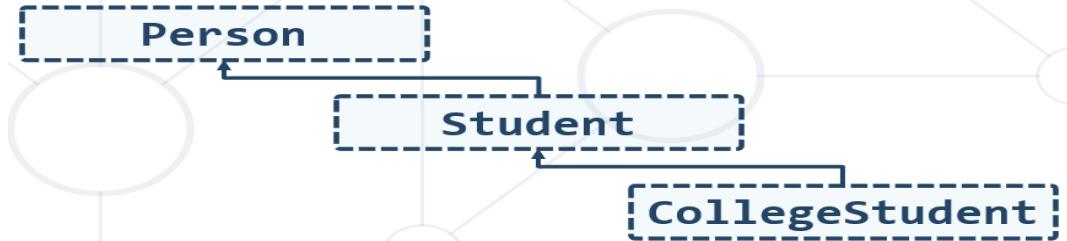
```
Student student = new Student();
student.sleep();
Employee employee = new Employee();
employee.sleep();
```

- 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);
}
}

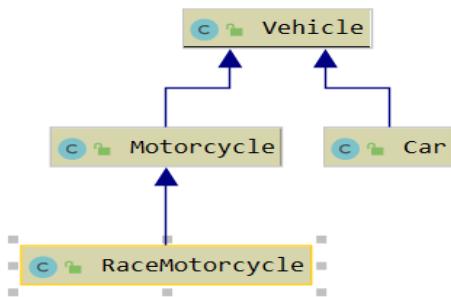
```

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.е. думичката super реферира само едно ниво нагоре – НЕ Е съвсем така – ако в междинния 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!



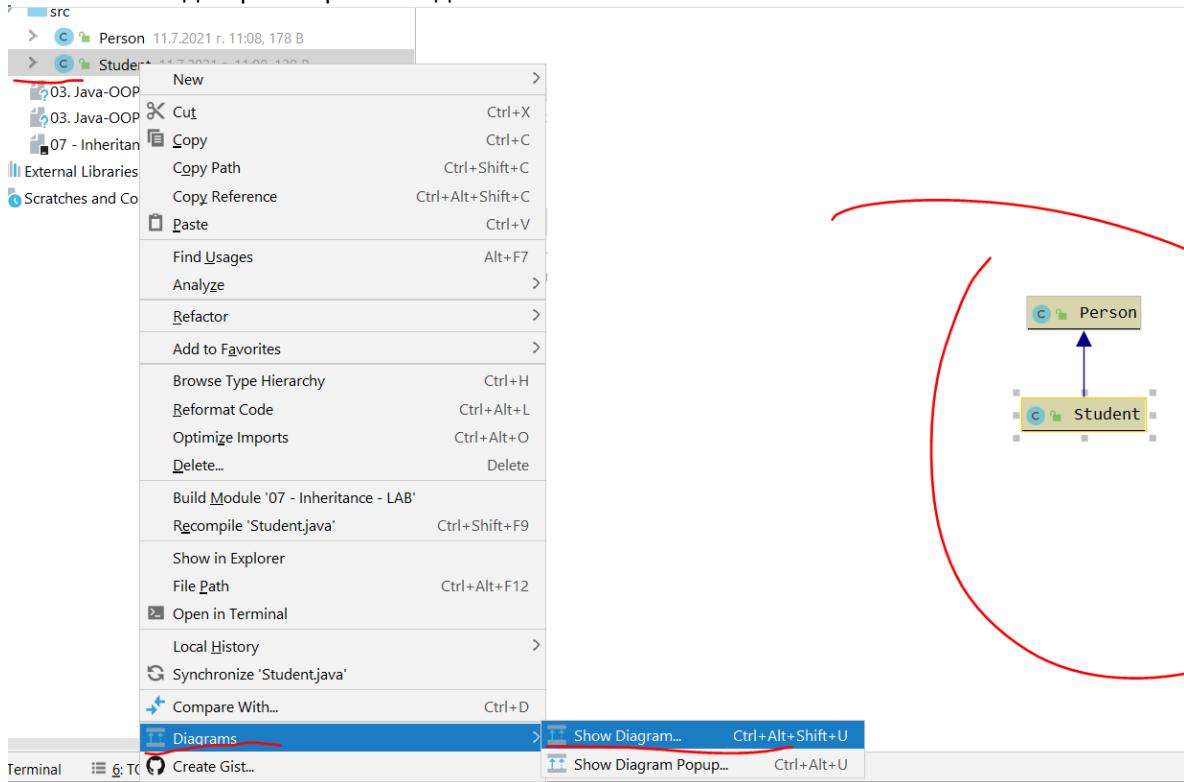
```

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



25.2. 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

25.3. 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();  
    }  
}
```

25.4. 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**

25.5. 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.



- показва, че класът е абстрактен

26. Encapsulation

26.1. 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 чак толкова много нещата, важното е че мога да го използвам
- Structural changes remain local – ако променим нещо отвън, то няма как да промени енкапсулираната 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
}
```

26.2. 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 снем/създадем обект
- при getter-а да върнем unmodifiable list

Т.е. за контрол и security моделиране.

26.3. Access Modifiers

private -> Package Private(Default) -> protected -> public

26.3.1. 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**

26.3.2. 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());
```

26.3.3. 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

26.3.4. 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**

26.4. 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());  
}
```

26.5. 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;

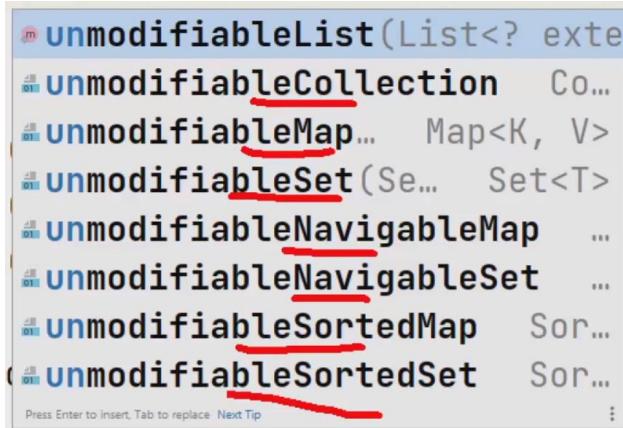
    public void addPlayer(Person person) {
        this.players.add(person); // Add new methods for functionality over list
    }

    public List<Person> getPlayers() {
        return Collections.unmodifiableList(players); // Returns a safe collections – на която не можем да добавяме
// элементи, 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



26.6. 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

Can we have a class that is both abstract and final in Java? – No. Because a final class can not be extended! (final abstract class would be changed depending who implements it, and the final keyword does not allow any change.)

27. Interfaces and Abstraction

27.1. 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 правя без детайлите
- Achieved with **interfaces** and **abstract classes**
- **Abstraction can not be instantiated**

Encapsulation

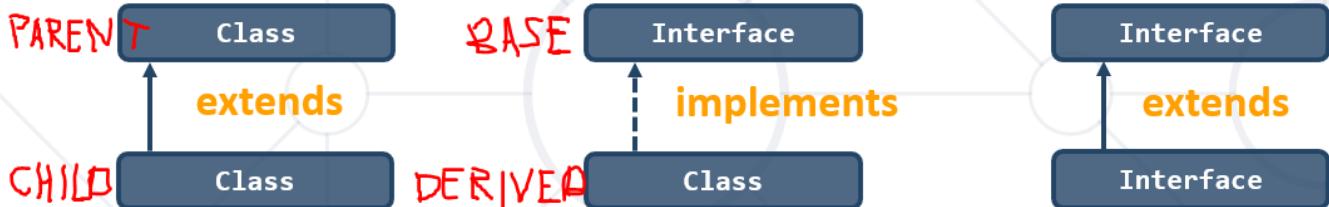
- Process of **wrapping code and data together into a single unit** – обвиване на код и данни в единица

- 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**

27.2. 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 читателите 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!!!
- Дафайлт-ния метод не искаме да го override-ваме в наследниците класове

In interface, a default method has an implementation, can not be overridden, and it is always executed

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);
}
```

27.3. 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(); - абстракният метод задължава наследника да имплементира
```

27.4. Interfaces vs Abstract Classes

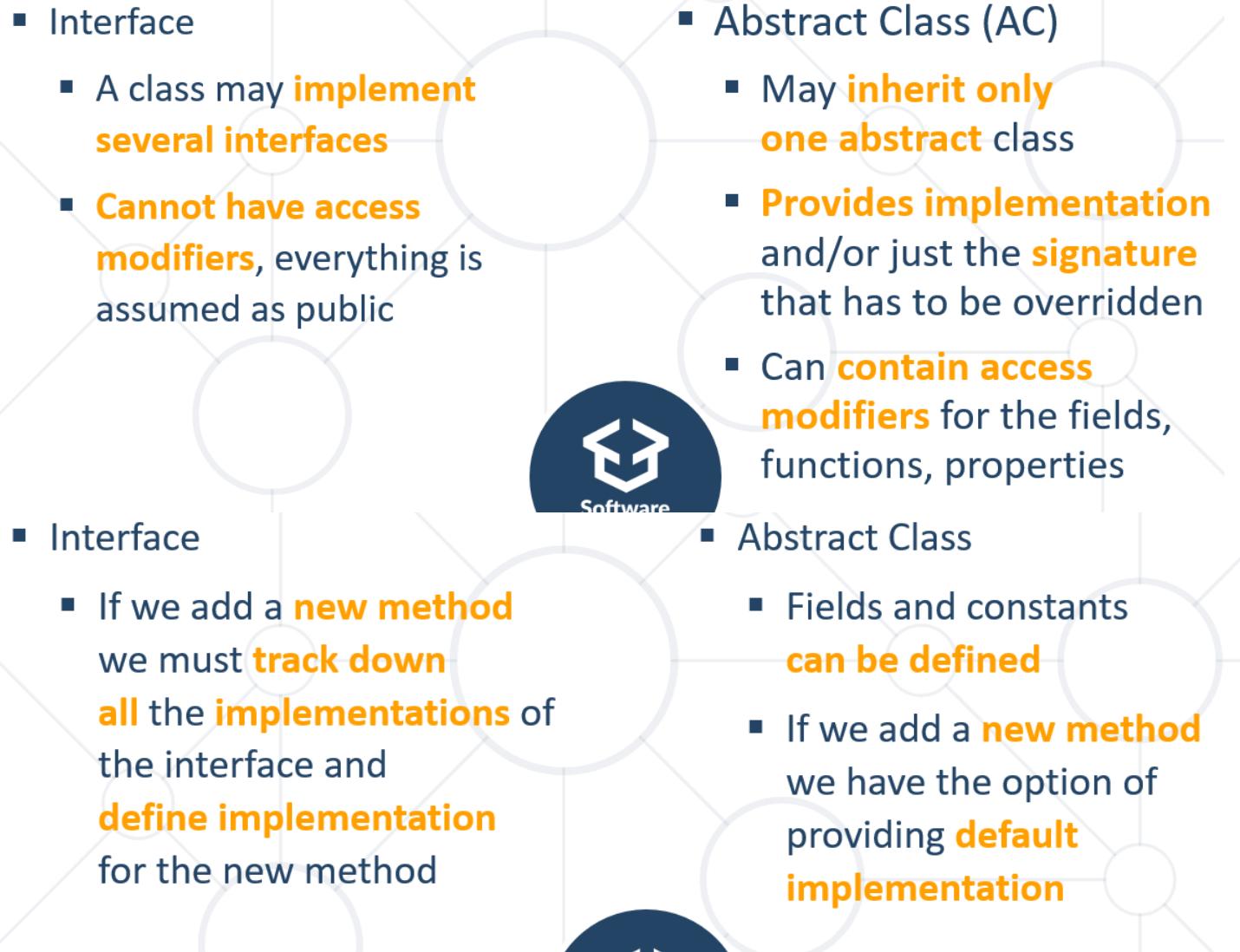
abstract class can have a constructor?

Yes, an **abstract class can have a constructor in Java**. You can either explicitly provide a constructor to the abstract class or if you don't, the compiler will add a default constructor of no argument in the abstract class.

An interface cannot contain a constructor.

Which of the following is a difference between an interface and an abstract class ?

- a.An abstract class can have a constructor while an interface cannot
- b.An interface can have default methods while an abstract class cannot
- c.An abstract class can **define** instance variables while an interface cannot
- d.**All of the above**



27.5. Example interfaces and reflection

```
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

```

28. Polymorphism

28.1. 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)

Референциятаказва 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

28.2. 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
    }
}

```

28.3. Types of Polymorphism

Which of the following is a difference between method overriding and method overloading?

- a.In method overriding the number of parameters must be the same while in method overloading it is not necessary
- b.all of the above
- c.In method overriding the access modified must be the same (or not more restrictive access modifier) while in method overloading it is not necessary
- d.In method overriding the return type must be the same while in method overloading it is not necessary

28.3.1. Runtime Dynamic polymorphism (overriding)– we use here method overriding

Different Ways to Prevent Method Overriding in Java

1. Using a static method.
2. Using private access modifier.
3. Using default access modifier.
4. Using the final keyword method.

```

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**

Is the following definition correct: yes

```

public interface Service {

    abstract public void invoke() throws Exception;
}

```

28.3.2. *Compile time Static polymorphism (overloading)* – we use here **method overloading**

Compile Time Polymorphism In Java is also known as **Static Polymorphism**. Furthermore, the call to the method is resolved at compile-time. Compile-Time polymorphism is achieved through **Method Overloading**. This type of polymorphism can also be achieved through **Operator Overloading**. However, Java does not support Operator Overloading.

Method Overloading is when a class has multiple methods with the same name, but the number, types, and order of parameters and the return type of the methods are different. Java allows the user freedom to use the same name for various functions **as long as it can distinguish between them by the type and number of parameters, but not by the return type!**

```

no usages  svilkata
private Long isItemIdANumber(String itemId) {
    final Long itemLongId;
    try {
        itemLongId = Long.parseLong(itemId);
    } catch (Exception e) {
        throw new ObjectIdNotANumberException(String.format("%s is not a valid laptop item number!", itemId));
    }
    return itemLongId;
}

💡 no usages  new*
private Integer isItemIdANumber(String itemId) {
    final Integer itemLongId;  'isItemIdANumber(String)' is already defined in 'bg.softuni.computerStore.service.LaptopService'
    try {
}

```

```

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**

28.4. 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

28.5. 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);
    }
}

```

28.6. 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);  
}
```

28.7. 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

29. SOLID = S.O.L.I.D.

Clean Architecture book: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series) 1st Edition

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
    }
}

```

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());
    }
}

```

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, като не изменят 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.width; // 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**

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 someting} //ok e
    public void sleep(Do something) //ok e
}

public class Robot implements Worker {
    public void work() {Do something} //ok e
    public void sleep() {
        throw new UnsupportedOperationException(); // не е ок
    }
}

```

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...
    }
}
```

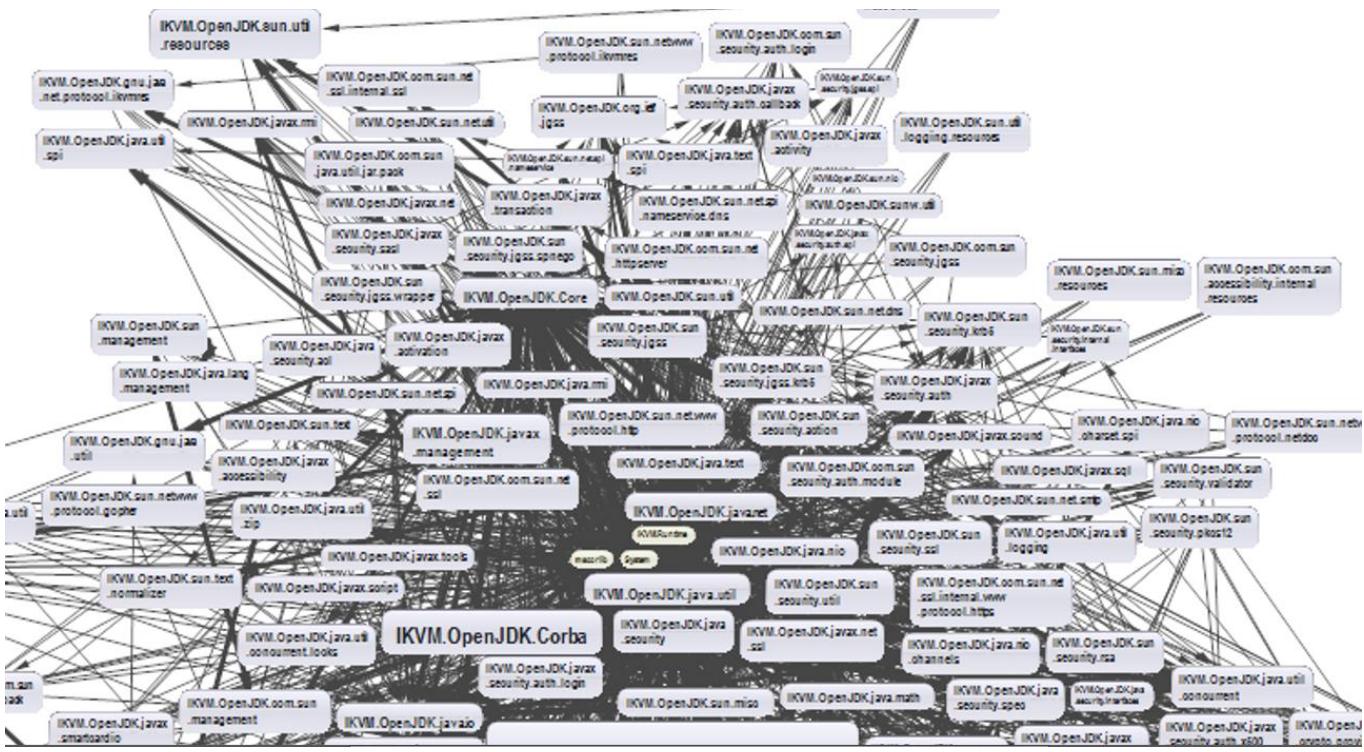
5. D – [Dependency inversion principle](#) – one should depend upon abstractions, not concretions

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

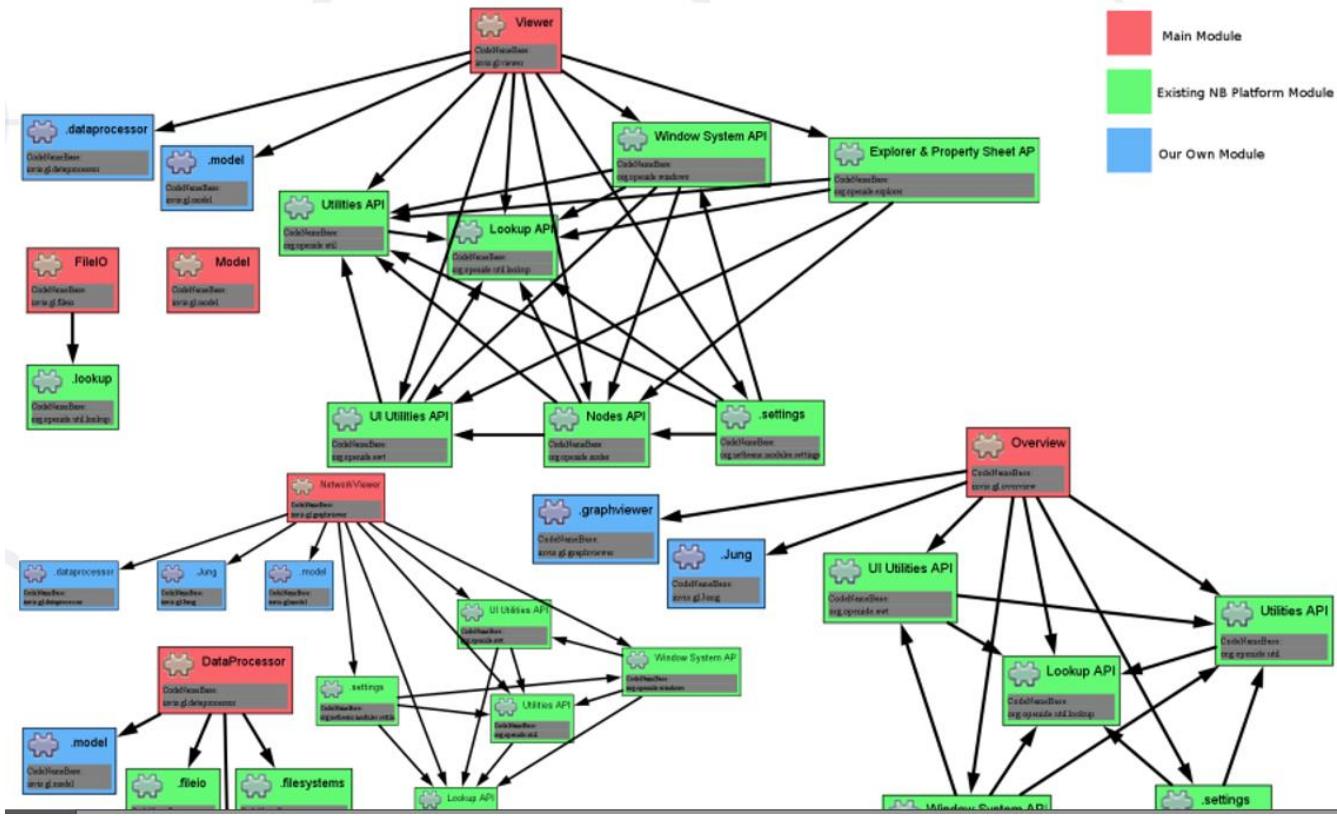
Ако в един interface има вложен клас, то това е в разрез с dependency inversion принципа.

What happens when modules **depend directly** on **other modules** – it becomes a mess



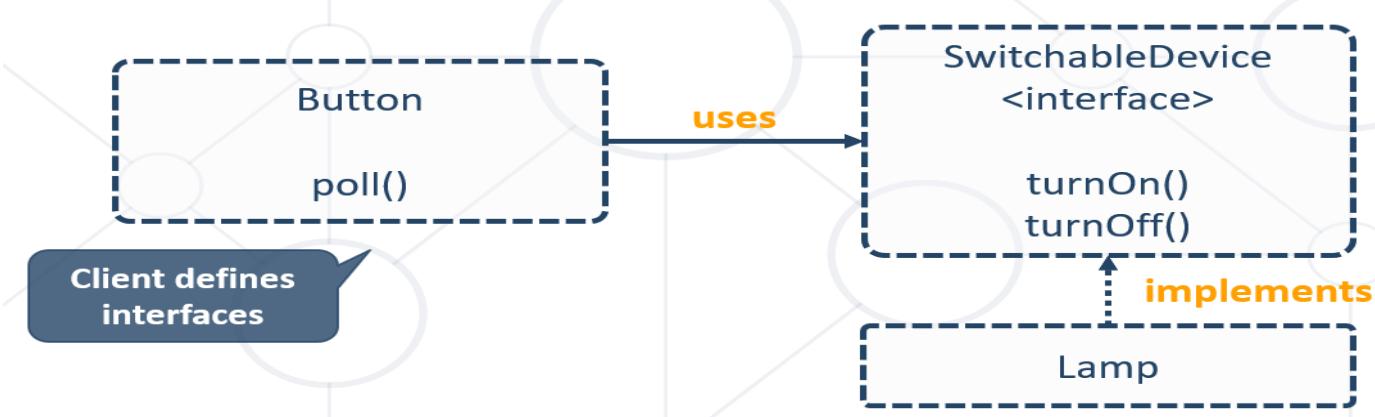
The goal is to depend on abstractions

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

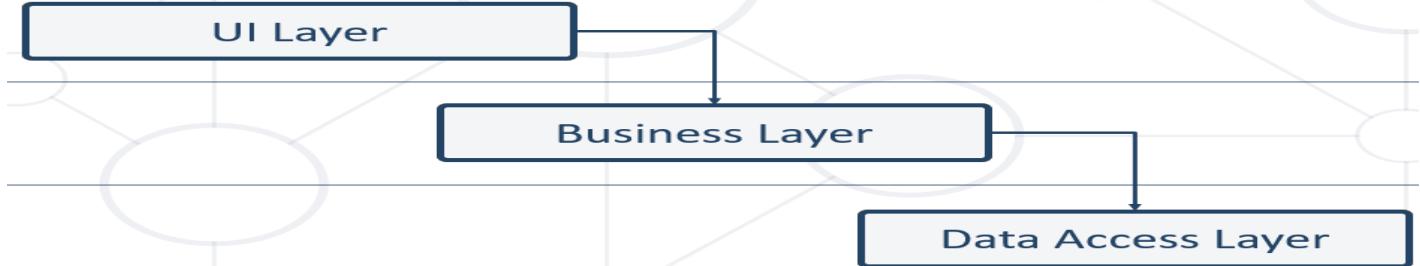
- Pros - No change in rest of the class; Very flexible

- **Cons** - Many parameters; Breaks the method signature

```
public class Copy {
    public void copyAll(Reader reader, Writer writer) {}
}
```

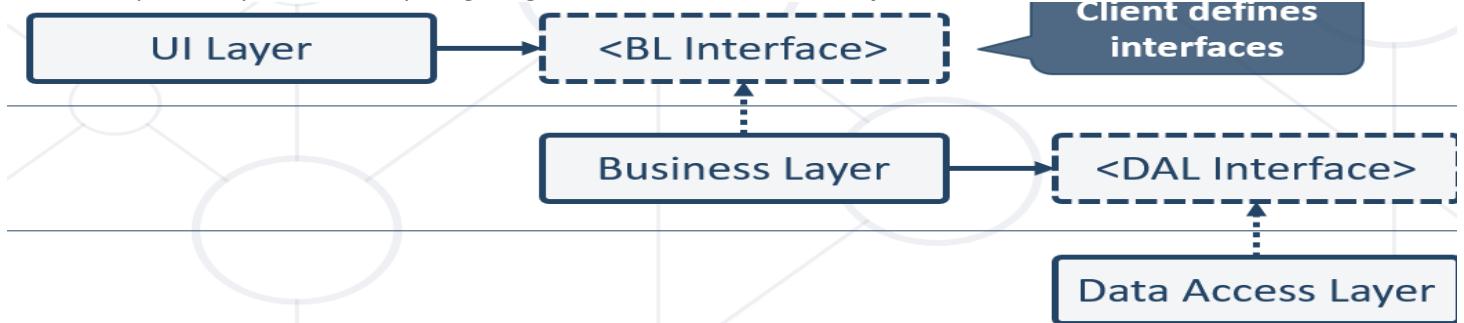
Layering (1)

- Traditional programming - **High-level** modules use **low-level** modules



Layering (2)

- Dependency Inversion Layering - High and low-level modules **depend on abstractions**



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();
        }
    }
}
```

30. Reflection and Annotations

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**

Reflection uses

In practice many frameworks and libraries make use of reflection for different purposes:

- Spring framework uses reflection to dynamically load beans from XML definitions
- JAXP (Java API for XML Parsing) to dynamically load the XML parser to use
- Persistence frameworks like Hibernate uses reflection to process Hibernate configuration and determine how to link entities
- Unit testing frameworks like JUnit use reflection to determine test methods in a class that need to be called

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.getSimpleName();
```

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);
}
```

Getting the class

- For an object, the class can be retrieved with the **getClass()** method

```
Table table = new Table();
Class<? extends Table> aClass = table.getClass();
```

- The class instance can be retrieving by using **.class** on the primitive or object type

```
Class<Integer> intClass = int.class;
Class<Table> tableClass = Table.class;
Class<double[][]> doubleMatrixClass = double[][] .class;
```

- The static **Class.forName()** method can also be used to retrieve the class instance from the fully qualified class name.

```
Class.forName("Banana"); //cannot be used for primitive types.
```

- For primitive types the corresponding class can also be retrieved through the wrapper type using the **TYPE** static field:

```
Class<Integer> intclass = Integer.TYPE;
Class<Double> doubleClass = Double.TYPE;
```

- A class can also be retrieved using any of the methods depending on the particular scenario:

<code>Class.getSuperclass()</code>	Returns the superclass Class instance
<code>Class.getClasses()</code>	Returns the public member classes and interface Class instances (including the inherited)
<code>Class.getDeclaredClasses()</code>	The same as <code>getClasses()</code> but returns only the classes declared in the class (with any package access modified)
<code>Class.getDeclaringClass()</code>	Returns the wrapper class where the specified is declared
<code>Class.getEnclosingClass()</code>	Returns the enclosing class (i.e. for anonymous type)
<code>Field.getDeclaringClass()</code>	Returns the class of an instance field
<code>Method.getDeclaringClass()</code>	Returns the class of an instance method
<code>Constructor.getDeclaringClass()</code>	Returns the class of a constructor



Class introspection

- The following methods from the Class API can be used to retrieve information about the class:

<code>getModifiers()</code>	Returns the class modifiers
<code>getField()</code>	Returns a class field by name
<code>getDeclaredField()</code>	
<code>getFields()</code>	Returns the class fields
<code>getDeclaredFields()</code>	
<code>getMethod()</code>	Returns a class method by name and parameter types
<code>getDeclaredMethod()</code>	
<code>getMethods()</code>	Returns the class methods
<code>getDeclaredMethods()</code>	
<code>getConstructor()</code>	Returns a constructor by parameter types
<code>getDeclaredConstructor()</code>	
<code>getConstructors()</code>	Returns the class constructors
<code>getDeclaredConstructors()</code>	

`getFields` връща всички полета, включително и скритите полета на super класа. Например на обекта от тип `String`, има скрити super полета на класа `Object`.

3. Constructors, Fields and Methods

Constructors

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 "Horsemman": unit = new Horsemman(); break;
//        case "Gunner": unit = new Gunner(); break;
//    }
}

```

Constructor manipulation

The following methods can be used on a Constructor instance:

<code>getParameters()</code>	Returns the constructor parameters
<code>getExceptionTypes()</code>	Returns the exceptions that the constructor declares using throws
<code>getModifiers()</code>	Returns the constructor modifiers
<code>newInstance()</code>	Creates a new object instance using the constructor and the expected parameters
<code>getDeclaredAnnotations()</code>	Returns the annotations declared on the method

A class with a default constructor can also be instantiated with the `Class.newInstance()` method but since JDK 9 it is deprecated and should not be used

Fields

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().getSimpleName(); // 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.
```

Field manipulation

The following methods can be used on a Field instance:

getType()	Returns the field type
getGenericType()	
getModifiers()	Returns the field modifiers
get() getInt()/getLong() ...	Returns the current value of the field. Getters values are available for fields or primitive type
set() setInt()/setLong() ...	Sets a value for the field on a specified instance. Setter methods are available for fields of primitive type
setAccessible()	It true allows a value of the field to be set even if private or final
getName()	Returns the field name
getDeclaredAnnotations()	Returns the annotations declared on the field

Since modifiers are returned as an integer representing the modifiers as flags the static **Modifier.toString()** method can be used to return the modifiers as a string

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(); //а инстанция на класа!!!
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
    // а blackBoxInt е инстанция на класа BlackBoxInt
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
}
```

Method manipulation

The following methods can be used on a Field instance:

getReturnType()	Returns the return type of the method
getGenericReturnType()	
getParameters()	Returns the method parameters
getExceptionTypes()	Returns the exceptions that the method declares using throws
getModifiers()	Returns the method modifiers
getName()	Returns the method name
invoke()	Returns the field name
getDeclaredAnnotations()	Returns the annotations declared on the method

Since JDK 8 parameter names can be retrieved if the **-parameters** option is passed to the Java compiler

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()));
}
```

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;  
}
```

6. Arrays and Java reflection

- The reflection API provides an `Array` class that can be used to create an array:

```
Table[] tables = (Table[]) Array.newInstance(Table.class, 10);
```

- The class also provides getter/setter methods that can be used to modify or retrieve a value from the array

```
Array.get(tables, 5);  
Array.get(tables, 2, new Table());
```

- 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();
```

7. Other class operations

Additional useful operations from the `Class` API include:

<code>isArray()</code>	Checks if the given class is an array
<code>isEnum()</code>	Checks if the given class is an enum
<code>getEnumConstants()</code>	Returns the enum constants (if the class is an enum)
<code>isInstance()</code>	Checks if a given object is instance of the class
<code>isAssignableFrom()</code>	Determines if the class is a superclass/superinterface of the class/interface provided as a parameter
<code>cast()</code>	Casts an object to the class or interface represented by the <code>this Class</code> instance
<code>getPackage()</code>	Returns the package of the class
<code>getModule()</code>	Returns the module of the class
<code>getInterfaces()</code>	Returns the <code>Class</code> instances of the interfaces implemented by the class

<code>getClassLoader()</code>	Returns the ClassLoader instance used to load the class
<code>getClasses()</code>	Returns the member classes of the class
<code>getAnnotations()</code>	Returns the annotations declared on the class
<code>getResource()</code>	Returns a URL to a resource (file) relative to the class
<code>getResourceAsStream()</code>	Returns an InputStream to a resource (file) relative to the class
<code>getProtectionDomain</code>	Returns the protection domain of the class

The protection domain of a class conveys the security-related information for the class as per the Java security sandbox model discussed later in the session

8. `setAccessible(true)` extra security

Вече не е достатъчно да кажем само `setAccessible(true)`, а трябва да подадем на Java виртуалната машина и допълнителни настройки, за да ни позволи, като тези примерно:



The screenshot shows an IDE interface with the following details:

- Build and run**: Shows the command `java 17 SDK of 'Sdk2Examples' module`.
- Modify options**: Shows the command `--add-opens java.base/java.util=ALL-UNNAMED --add-opens java.base/java.lang=ALL-UNNAMED`.
- Code Editor**:


```
public class ReflectionDemo {
    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
        Field[] declaredFields = String.class.getDeclaredFields();
        for (Field declaredField : declaredFields) {
            System.out.println(declaredField);
        }

        String hello = "Hello!";
        Field valueField = String.class.getDeclaredField("value"); //има такова скрито поле
        в обекта String. Все още не знаем този Field за кой точно обект е
        valueField.setAccessible(true);

        byte[] value = (byte[]) valueField.get(hello /* this */); //hello.value
        System.out.println(Arrays.toString(value));
    }
}
```

9. Примерно демо с reflection

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

```

//           System.out.println(numberOfInstanceFields(Integer.valueOf(100)));
//           System.out.println(numberOfInstanceFields(new java.util.LinkedHashMap<>()));
//           System.out.println(numberOfInstanceFields(new int[]{}));
//           System.out.println(numberOfInstanceFields(new
java.io.ByteArrayOutputStream()));
System.out.println(numberOfInstanceFields((AbstractMap) new TreeMap()));

}

// Task: return the number of all not-static fields (including the inherited ones
// directly) contained in the given object.
public static int numberOfInstanceFields(Object object) {
    java.lang.Class<?> toCheck = object.getClass();

    int[] count = new int[1];

    return travelAllPossibleClasses(toCheck, count);
}

private static int travelAllPossibleClasses(java.lang.Class<?> toCheck, int[] count) {
    if (toCheck == null) {
        return 0;
    }

    // directly inherited (extends only)
    Class<?> superclass = toCheck.getSuperclass();
    travelAllPossibleClasses(superclass, count);

    // not directly inherited (finds interfaces implemented by a class or extended by
another interface)
    Class<?>[] superInterfaces = toCheck.getInterfaces();
    for (Class<?> anInterface : superInterfaces) {
        travelAllPossibleClasses(anInterface, count);
    }

    return calculatePerClass(toCheck, count);
}

private static int calculatePerClass(Class<?> classToCheck, int[] count) {
    java.lang.reflect.Field[] declaredFields = classToCheck.getDeclaredFields();
    int currentClassCount = 0;
    for (java.lang.reflect.Field declaredField : declaredFields) {
        if (!java.lang.reflect.Modifier.isStatic(declaredField.getModifiers())) {
            count[0]++;
            currentClassCount++;
        }
    }

    System.out.println(classToCheck.getName() + ": " + currentClassCount);

    return count[0];
}
}

```

10. Dynamic proxies

Така наречения proxy клас, който ни дава възможност да дефинираме динамично обекти, които наследяват даден интерфейс
Обикновено за това си има готови библиотеки, които се справят по-добре отколкото ако ние тръгнем да си пишем тези проксита

- Dynamic proxies provide the possibility to tunnel method invocations of a class through a common method of a proxy class
- In the Java reflection class a dynamic proxy is created by the **java.lang.reflect.Proxy** class
- It provides the possibility to return a proxy that implements dynamically an interface
- The methods of the interface are executed by the **invoke()** method of a handler class

```

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.List;

public class DynamicProxy {
    public class DynamicInvocationHandler implements InvocationHandler {

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            //do some logic here when calling this method
            return null;
        }
    }

    //динамично да си направим обект/да декорираме обект от тип Лист интерфейс
    public static void main(String[] args) {
        Proxy.newProxyInstance(
            DynamicProxyTest.class.getClassLoader(),
            new Class[] {List.class},
            new DynamicInvocationHandler()
        );
    }
}

```

Dynamic proxies have a wide range of applications:

- Creation of database transactions (query is proxied to a method that commits or roll-back the transaction)
- Unit testing (for invoking mock(fake) objects instead of concrete implementations)
- For AOP (aspect-oriented-programming) whereby a method invocation can be intercepted and amended with additional logic or modified

11. Limitations of reflections

Although the Reflection API is quite powerful, it is still limited in terms of modifying the actual class (adding, modifying or deleting methods or fields, etc.). This capability can be provided by:

- Custom classloaders where every time a new class is required a new classloader instances in order an object of that class to be created/instantiated
- Provide a Java instrumentation agent using the Java instrumentation API

The reflection API also does not provide the capability to generate a Java class at runtime and load it in the JVM. This capability can be achieved by third-party libraries such as:

- **cglib**
- **asm**
- **Javassist**
- **BCEL**

The JShell API introduced in JDK 9 can also be used to dynamically create and load a Java class.

12. 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` – останяло/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

```
@SuppressWarnings(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 излизящи 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 анотация
        .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 анотация, или анотация Entity се използва само на един клас за момента
        .collect(Collectors.toList());

    return collect;
}
```

Първо анотиране в случая

```
1 annotationsByType = {Column[1]@2574}
2   0 = {$Proxy3@2580} "@annotations.Column(name="id")"
3     f h = {AnnotationInvocationHandler@2584}
4       f type = {Class@2568} "interface annotations.Column"... Navigat...
5         f memberValues = {LinkedHashMap@2585} size = 1
6           "name" -> "id"
7           f 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
```

13. 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();
    }
}

```

31. Exception Handling

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== verified) by the compiler at compilation-time - also called as Compile Time exceptions.
Верифицира се като се поставя или **throws** на дадения метод или като се хване с **try-catch** конструкция.

Реално ако Java компилатора не работи както трябва и пропусне верификацията, то системата би гръмнала at the time of execution. Т.е. всяка грешка, която гърми - винаги е по време на изпълнение!!!

Java компилатора обаче ще работи както трябва и ще изпиши при компилация към bytecode-а - че не сме сложили throws на даден метод. Т.е. компилацията няма да е успешна и ще каже/ще изплюе какво не е наред. И няма да стигнем до execution-а. Нещо повече – в IntelliJ има явно настройка, която предварително ни известява, и даже няма да тръгне да се компилира, а IDE-то ще ни подчертава кое не е наред.

В Java са решили само малко на брой грешки да бъдат причислени като checked – само тези грешки, от които програмата/application-а може лесно да се възстанови.

```
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.

Unchecked

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

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.

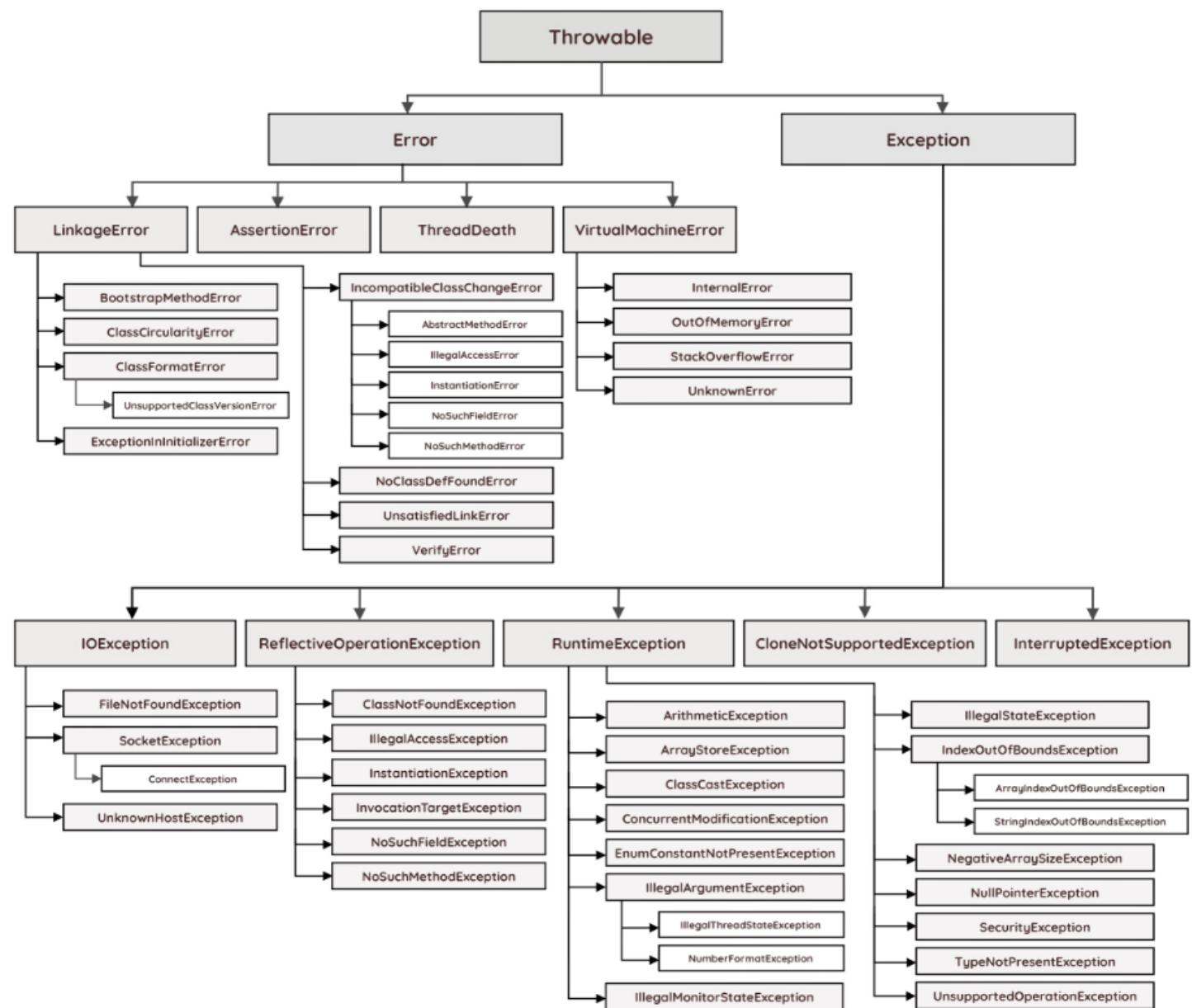
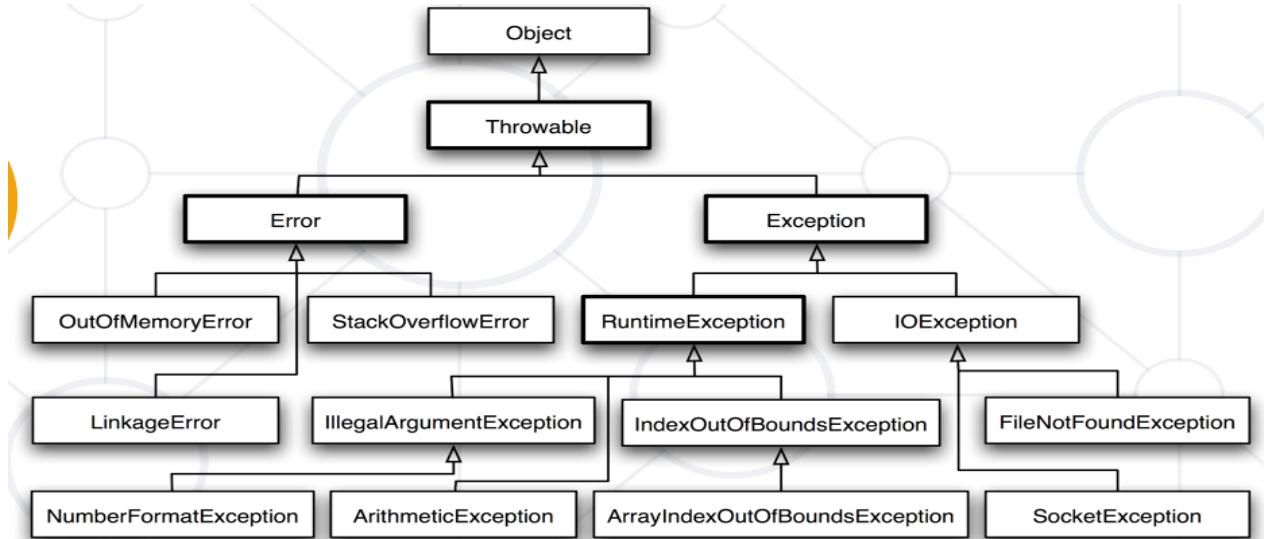
Error– an exception that occurs at the time of execution

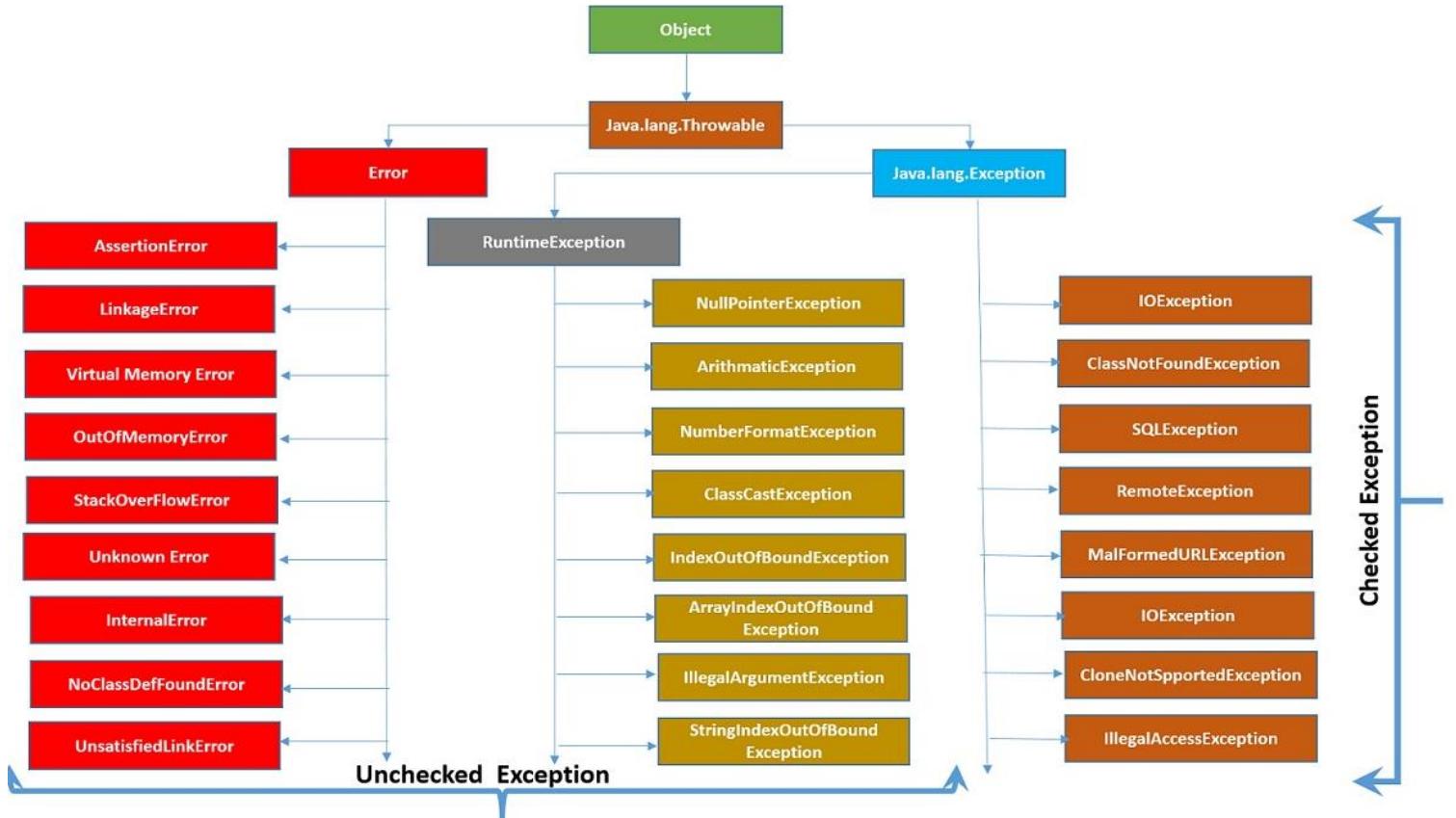
The second kind of exception is the **error**. These are exceptional conditions that are **external to the application**, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because 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 Examples





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.

2. Handling Exceptions

Some colleagues consider all the checked exceptions also to be handled exceptions 😊
 But we can handle actually all types of exceptions – both checked and unchecked.

How Programmer Handle an Exception?

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work.

- Program statements that you think can raise exceptions are contained within a **try** block. If an exception occurs within the try block, it is thrown. Your code can **catch** this exception (using catch block) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**. (**throw new Exception();**) Cannot throw multiple exceptions
- Any exception that is thrown out of a method must be specified as such by a **throws** clause. **Used to indicate what exception type may be thrown by a method**, Can declare multiple exceptions. Ако не се обработи в по-

горен слой с try-catch, то във всеки метод от по-горен слой също добавяме throws. Като накрая JVM обработва public static void main метода, на който също сме декларирали throws.

- Any code that absolutely must be executed after a try block completes is put in a **finally** block.

- 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  
}
```

- In Java exceptions can also be handled by the **try-catch** and with the help of the **throws** at the end of the method signature

```
public class TestThrows {  
    //defining a method
```

```

public static int divideNum(int m, int n) throws ArithmeticException {
    int div = m / n;
    return div;
}

//main method
public static void main(String[] args) {
    TestThrows obj = new TestThrows();
    try {
        System.out.println(obj.divideNum(45, 0));
    }
    catch (ArithmeticException e){
        System.out.println("\nNumber cannot be divided by 0");
    }

    System.out.println("Rest of the code..");
}
}

```

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();
}
```

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

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);  
    }  
}
```

Конвенция в java за игнориране на хванат exception

```
catch(SocketTimeoutException ignored){  
    ;  
}
```

32. Debugging

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

2. IntelliJ IDEA Debugger

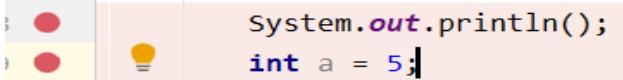
В режим на дебъгване, се зареждат куп допълнителни неща, които в нормалния режим на работа на кода, няма да се заредят.

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

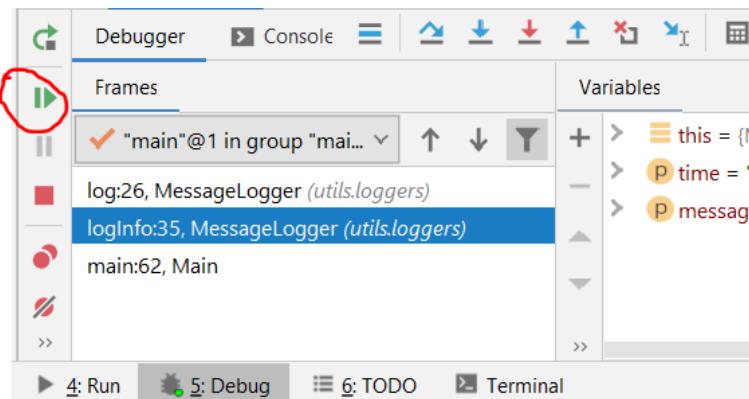
Adding **breakpoints**

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

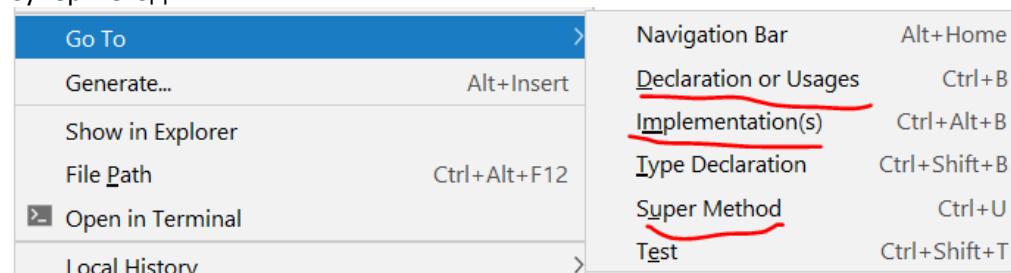
Да слагаме **breakpoint** на определените места

Data tips

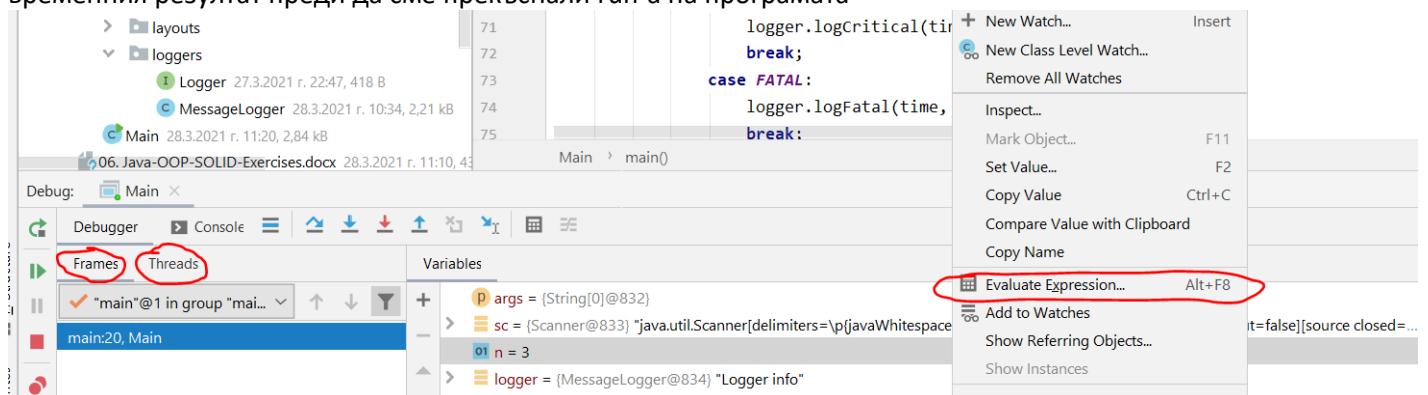
Декларации и употреби

Имплементации

Супер метод

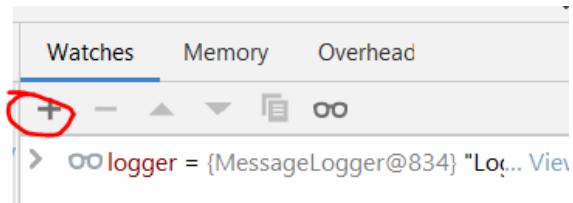
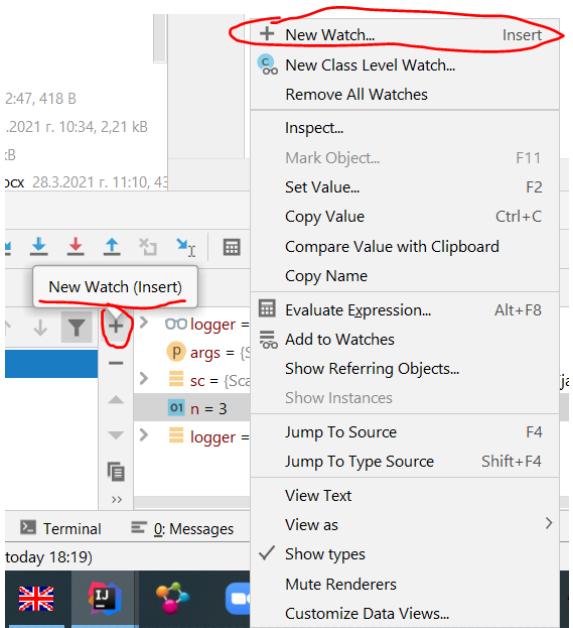


Evaluate expression – казва какво ще ретърне навън още преди да го е ретърнало – можем да го сравним с временния резултат преди да сме прекъснали run-а на програмата

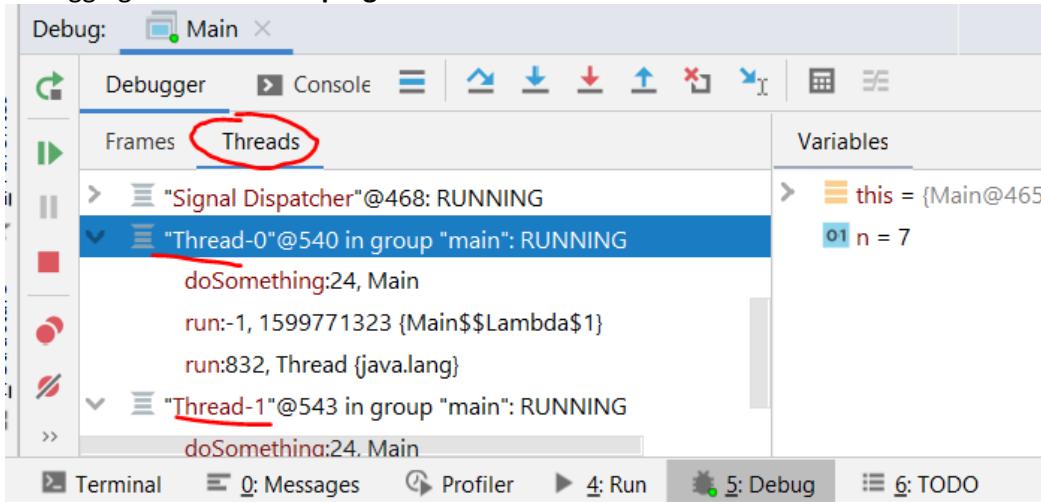


Watch variables

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**



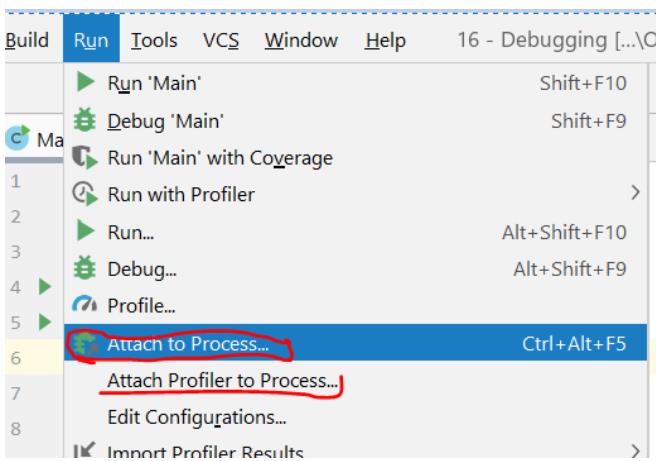
Debugging multithreaded programs



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 lines: one at the start of the run() method and another at the beginning of the while loop. The status bar indicates the application is running. Below the code editor is the 'Variables' tool window, which lists the current context and the value of 'this'. A red box highlights the 'Run to Cursor' button in the toolbar, with the keyboard shortcut 'Alt+F9' written next to it.

- 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)

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:

The screenshot shows the IntelliJ IDEA interface with the 'Breakpoints' tool window open. In the code editor, a breakpoint is set at line 58 of Main.java. The 'Condition:' field in the breakpoint dialog is highlighted with a red circle, containing the expression `reportLevel.equals(ReportLevel.CRITICAL)`. The code snippet below shows the execution path:

```

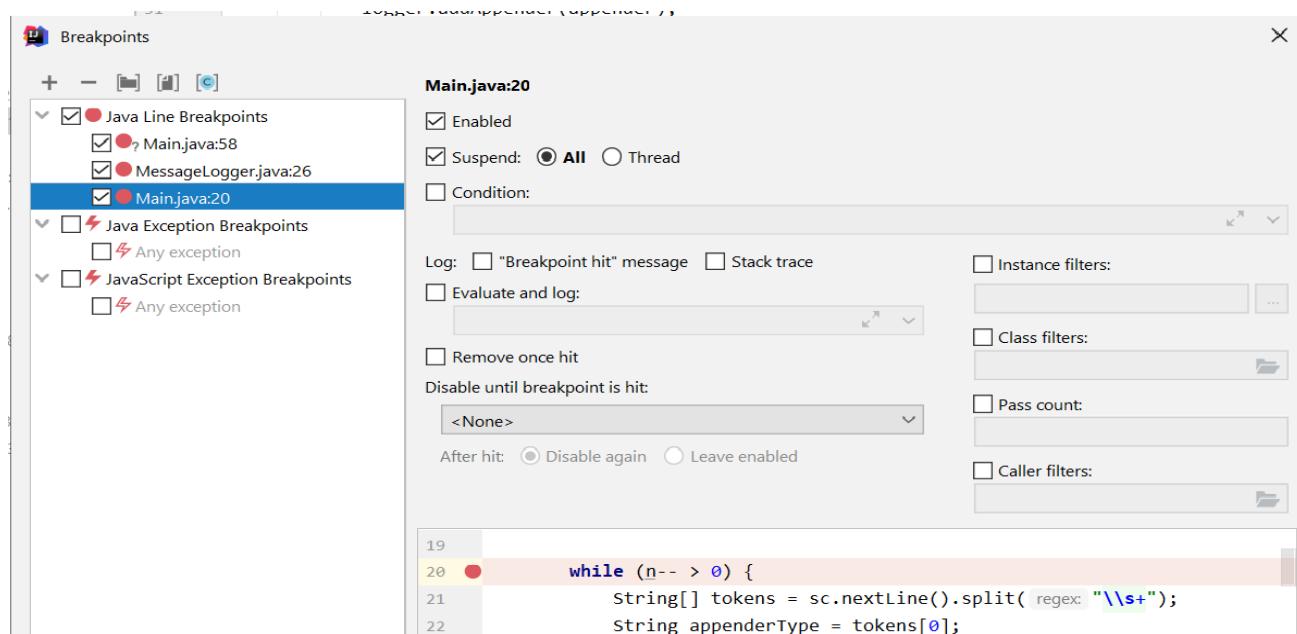
while (! input.startsWith("quit")) {
    String[] tokens = input.split( regex: "\\\\" );
    ReportLevel reportLevel = ReportLevel.valueOf(tokens[0]);
    String time = tokens[1];
    String appenderType = tokens[2];
    if (reportLevel == ReportLevel.INFO) {
        logger.logInfo(time, message);
    } else if (reportLevel == ReportLevel.WARN) {
        logger.LogWarning(time, message);
    } else if (reportLevel == ReportLevel.ERROR) {
        logger.LogError(time, message);
        break;
    } case CRITICAL:
        logger.logCritical(time, message);
        break;
    case FATAL:
        logger.logFatal(time, message);
        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

4. Stream debugging / Дебъгване на stream



Trace current stream chain

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

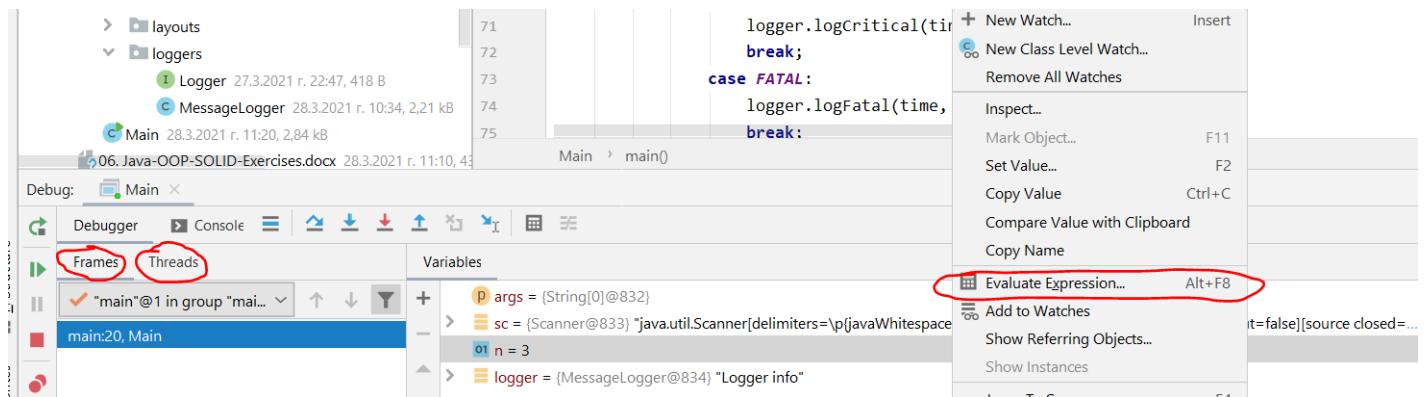
Right click on the variable and select "Add to Watches"

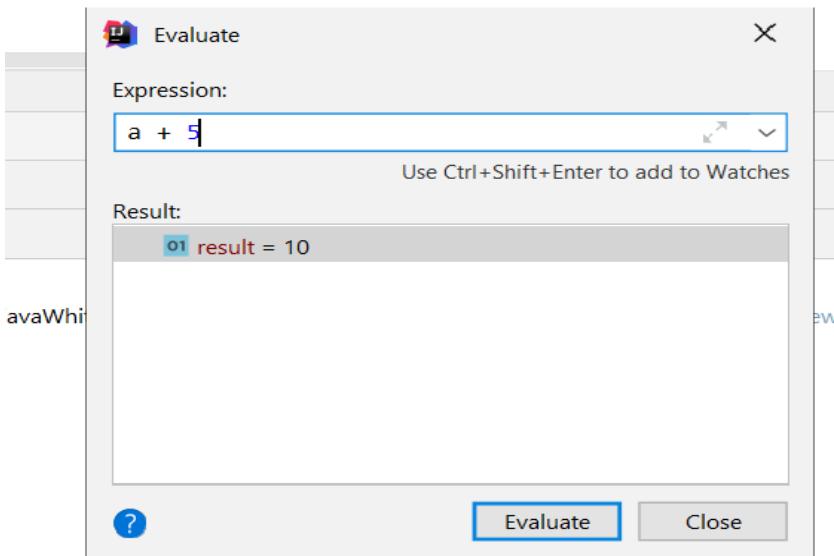
Write the variable name in Watches window

Evaluate Expression Window - казахме го по-горе

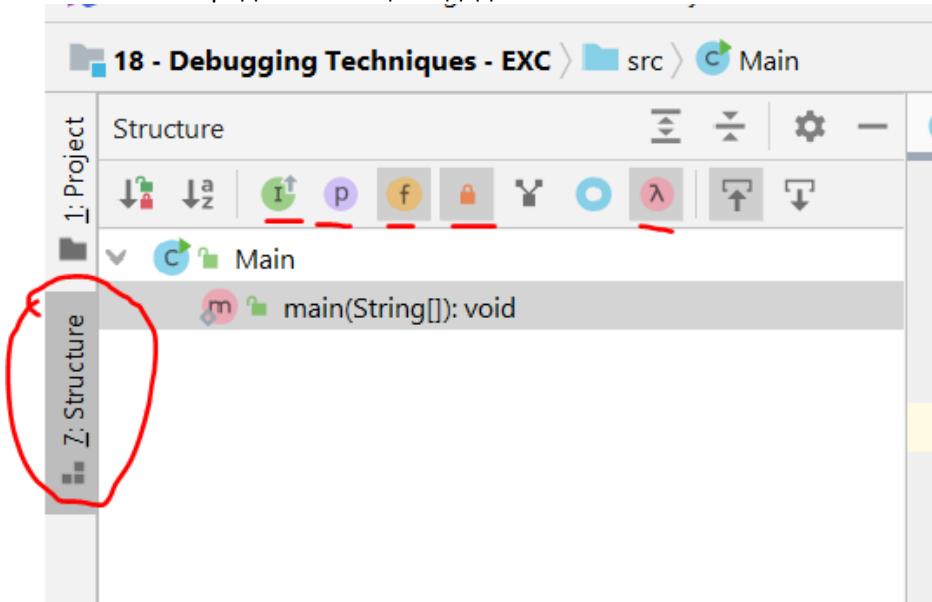
- Enables to evaluate expressions and code fragments in the context of a stack frame
- Also evaluate operator expressions, lambda expressions, and anonymous classes
- Shortcut – Alt + F8

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:



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

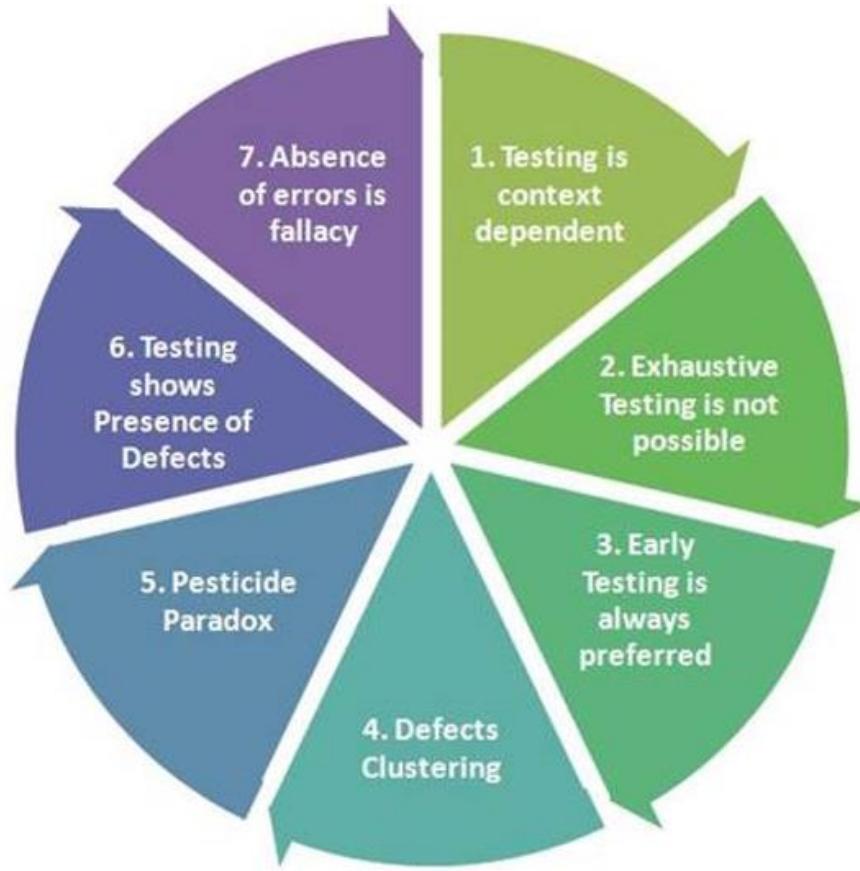
- Understand the problem before you fix it
- Understand the program, not just the problem
- Confirm the defect diagnosis
- Relax
- Save the original source code
- Fix the problem, not the symptom
- Make one change at a time
- Add a unit test that expose the defect
- Look for similar defects

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

33. Unit Testing

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

2. What is Unit Testing

What should be the "unit" under test in a JUnit test? - a public method from the main code we are only testing.

We do not test private, protected etc. methods.

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**

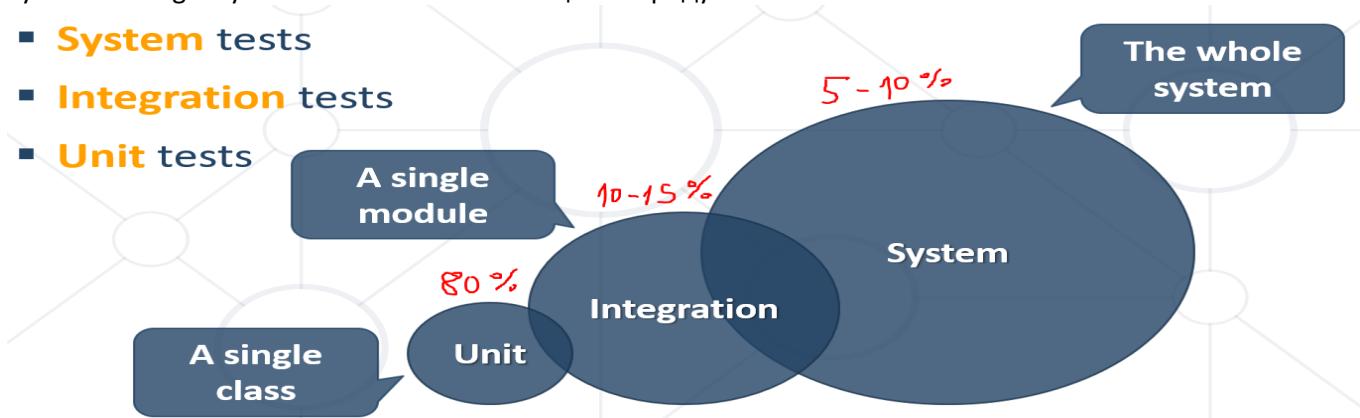
Automated Testing

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**



3. JUnit framework в Java

Install it

- Maven Repository (<https://mvnrepository.com/>) – всички неща, които има в Maven pom.xml

```

<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

```

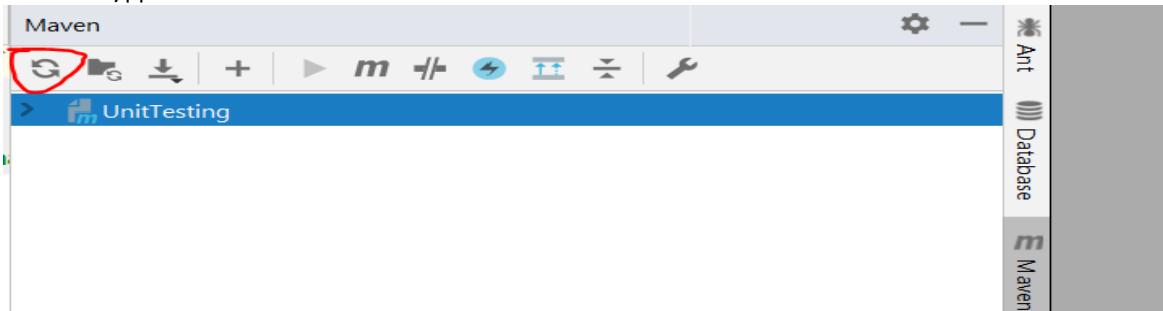
<project ...>
<groupId>softuni</groupId>
<artifactId>junit-example</artifactId>
<version>1.0-SNAPSHOT</version>
...
<properties>
  <maven.compiler.source> 13 </maven.compiler.source>
  <maven.compiler.target> 13 </maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>

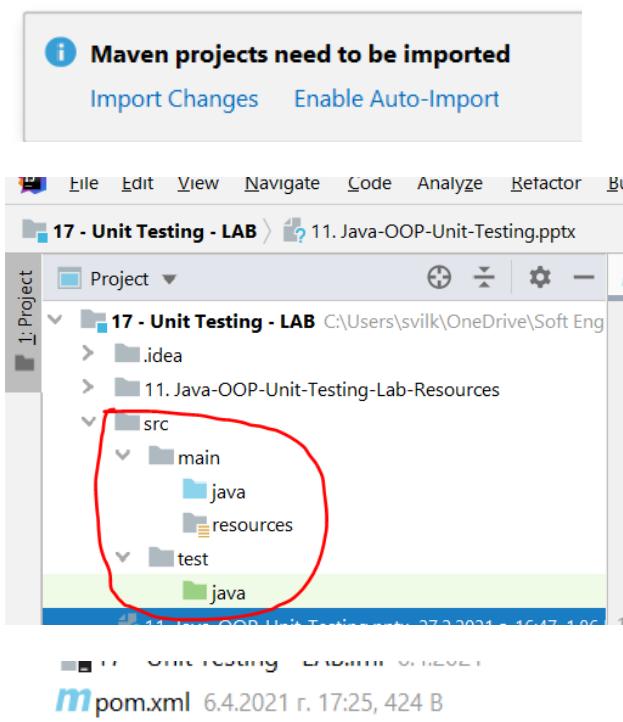
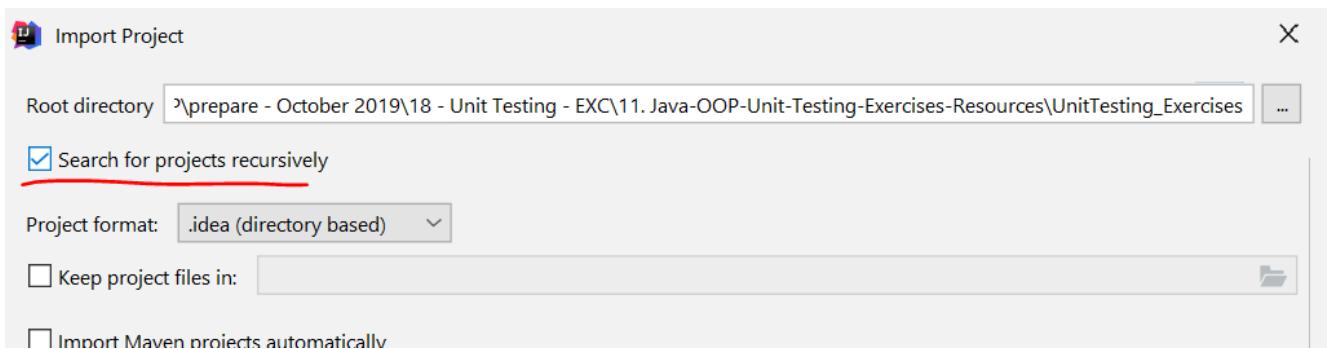
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Re-import All Maven Projects – когато добавим ново dependency в pom.xml файла, от тук може да го обновим/добавим.

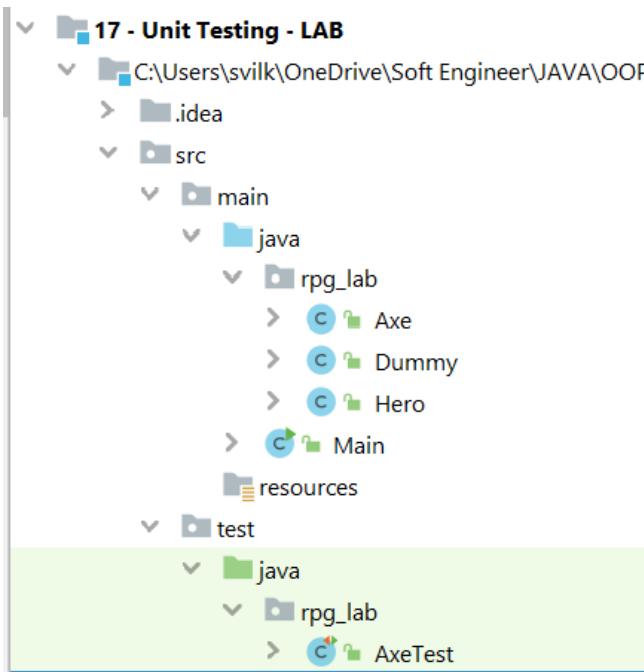


Намира проекти в директории надолу



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 (mpu eý) 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

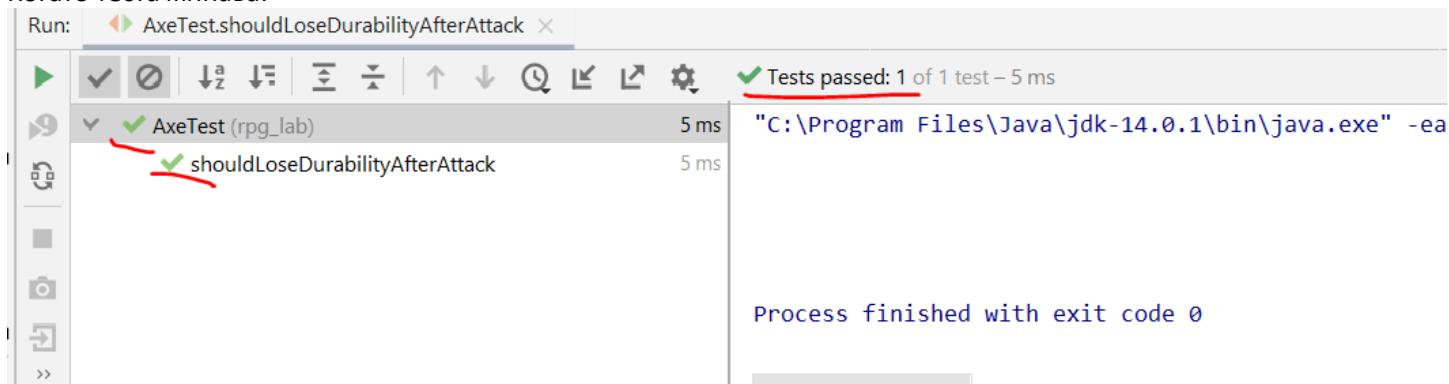
С нашите библиотеки, тук няма нужда да слагаме Assert. Но по принцип си го правим с **Assert.Throws()**

- Sometimes throwing an exception is the **expected behavior**

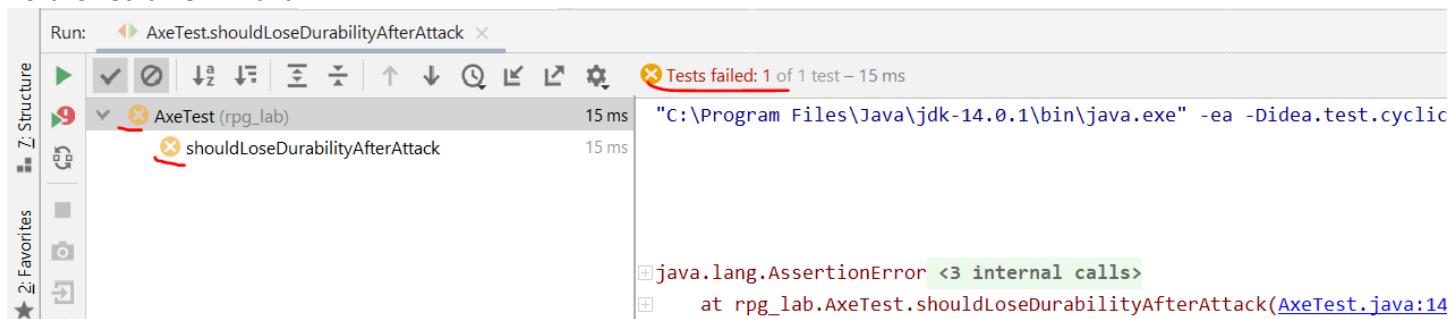
```
@Test(expected = IllegalArgumentException.class) //Assert
public void depositNegativeShouldNotAddMoney() {
    BankAccount account = new BankAccount(); //Arrange
    account.deposit(-50); //Act
}

Assert.assertTrue(axe.getDurabilityPoints() == 4);
```

Когато теста минава:



Когато теста не минава:



Example test pure JUnit5 when without Maven

```
class CollectionUtilsTest {

    @org.junit.jupiter.api.Test
    public void testNumberNotInTheList() {
        org.junit.jupiter.api.Assertions.assertFalse(
CollectionUtils.search(java.util.List.of(1, 2, 3), 5));
    }
}
```

```

@org.junit.jupiter.api.Test
public void testNumberInTheList() {
    org.junit.jupiter.api.Assertions.assertTrue(
CollectionUtils.search(java.util.List.of(1, 2, 3), 2));
}

@org.junit.jupiter.api.Test(expected = RuntimeException.class) to work it needs Maven
with a specific library imported
public void testNumberInTheListWhenListIsNull() {
    org.junit.jupiter.api.Assertions.assertThrows(RuntimeException.class, () ->
CollectionUtils.search(null, 2));
}

@org.junit.jupiter.api.Test
public void testNumberInTheListWhenNumberIsNull() {
    org.junit.jupiter.api.Assertions.assertThrows(RuntimeException.class, () ->
CollectionUtils.search(java.util.List.of(1, 2, 3), null));
}

class CollectionUtils {
    public static boolean search(java.util.List<Integer> numbers, Integer number) {
        try {
            return numbers.contains(number);
        } catch (NullPointerException npe){
            throw new RuntimeException();
        }
    }
}

```

Scenarios to test

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.

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 зануяваме 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

- 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.

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);  
}
```

@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

5. Dependencies

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;
    }
}
```

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  
}
```

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...  
}
```

5.4. Mocking

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/проверява/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 testAlarmClockShouldRingInTheMourning() {  
    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 врни true  
  
    Mockito.when(sensor.popNextPressurePsiValue()).thenThrow(IllegalArgumentException.class);  
    //когато по-надолу ти се извика метод popNextPressurePsiValue(), то врни exception
```

```
AlarmClock clock = new AlarmClock(mockedTime);
```

```

if (mockedTime.isMorning()) { //реално винаги е true
    Assert.assertTrue(clock.isRingding());
}
}

```

- 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
//класса 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!!!**

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);
    }
}

```

8. Other libraries in Maven

Досега видяхме JUnit и Mockito.

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
```

9. More

Which of the following **is not** a feature of the JUnit framework?

- a.repeated tests
- b.parameterized tests
- c.dynamic tests
- d. test generation

Repeated tests

Each execution of the `@RepeatedTest` will behave like a regular `@Test` having full JUnit test life cycle support. Meaning that, during each execution, the `@BeforeEach` and `@AfterEach` methods will be called.

```
@RepeatedTest(3)
void repeatedTest(TestInfo testInfo) {
    System.out.println("Executing repeated test");

    assertEquals(2, Math.addExact(1, 1), "1 + 1 should equal 2");
}
```

Parameterized tests

See my other notes

Dynamic tests

JUnit 5's Dynamic Tests feature, which allows us to declare and run test cases generated at run-time. Contrary to Static Tests, which define a fixed number of test cases at the compile time, Dynamic Tests allow us to define the test cases dynamically in the runtime.

Dynamic tests can be generated by a factory method annotated with `@TestFactory`. Let's have a look at the code:

```
@TestFactory
Stream<DynamicTest> translateDynamicTestsFromStream() {
    return in.stream()
        .map(word ->
            DynamicTest.dynamicTest("Test translate " + word, () -> {
                int id = in.indexOf(word);
                assertEquals(out.get(id), translate(word));
            })
        );
}
```

This example is very straightforward and easy to understand. We want to translate words using two `ArrayList`, named `in` and `out`, respectively. The factory method must return a `Stream`, `Collection`, `Iterable`, or `Iterator`. In our case, we chose a Java 8 `Stream`.

Please note that `@TestFactory` methods must not be private or static. The number of tests is dynamic, and it depends on the `ArrayList` size.

@Disabled

Which annotation can you use in JUnit 5 to disable a test? - `@Disabled`

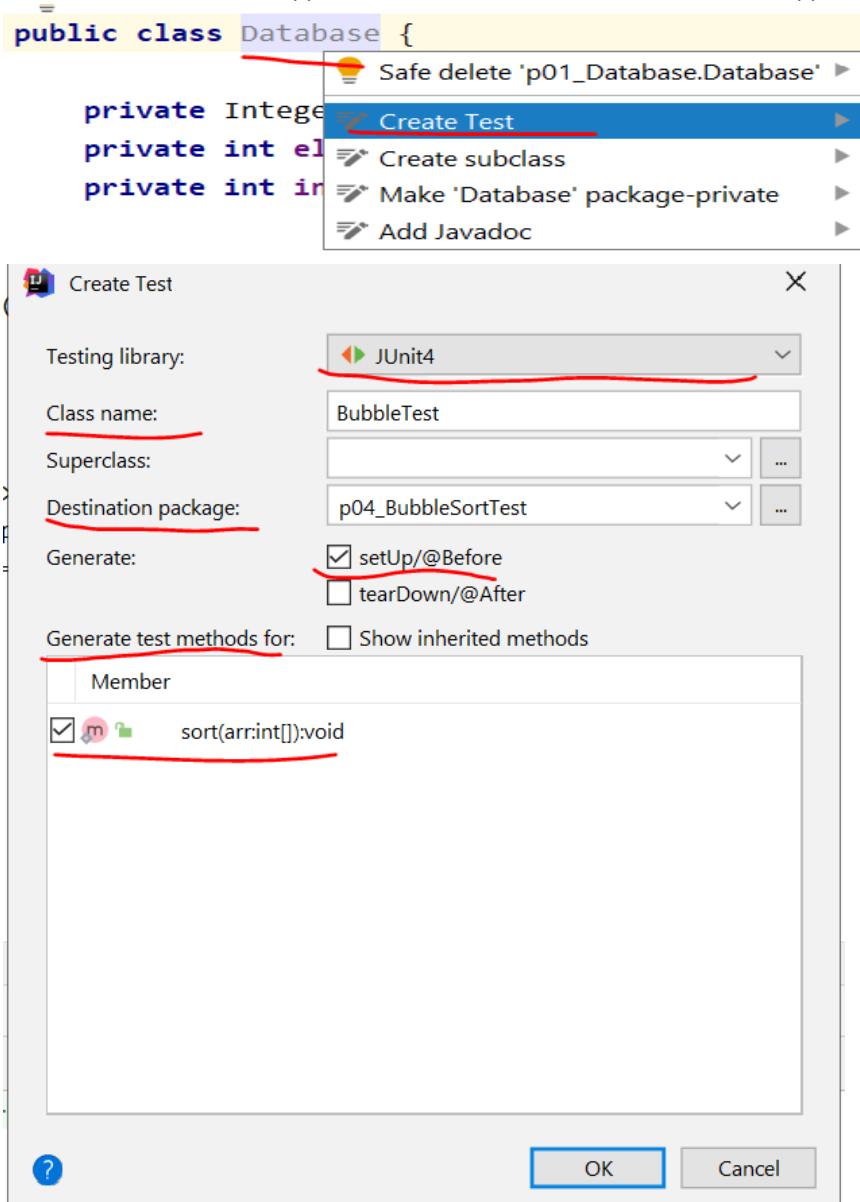
Assertions

Which of the following **is not a valid assertion** in JUnit API?

- a.assertSame() - Assert that expected and actual refer to the same object.
- b.assertArrayEquals - Assert that expected and actual arrays are equal.
- c.assertTimeout - Assert that execution of the supplied executable completes before the given timeout is exceeded.
- d.assertListEquals

10. 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);
}
```

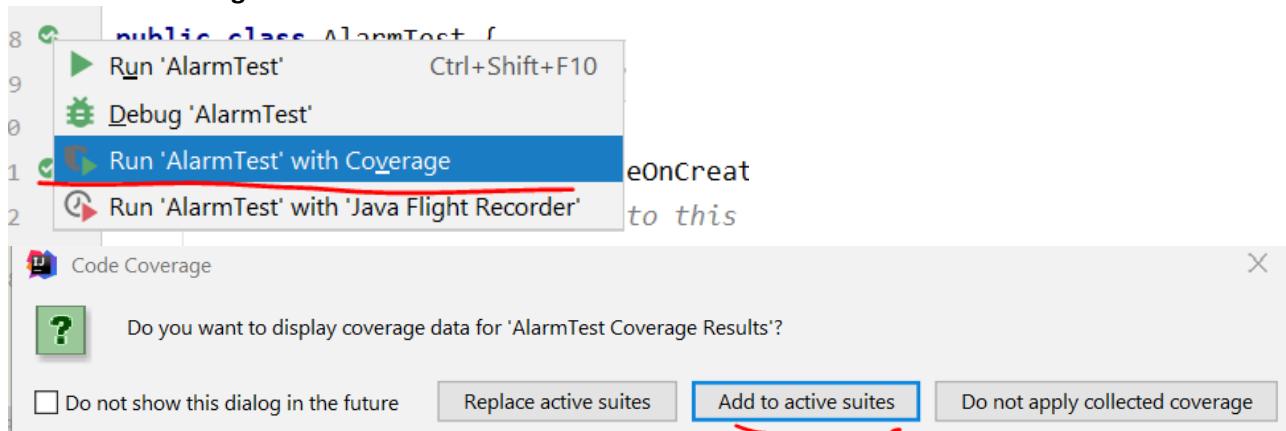
Проверка дали два масива са еднакви с еднаква подредба – къс вариант

```
@Test
public void databaseCreationTestShouldSetElementsInCorrectOrderAccordingToInitialParameters() {
    Integer[] elements = this.database.getElements();

    Assert.assertArrayEquals(numbers, elements);
}

assertNotNull(field);
assertFalse(alarm.getAlarmOn());
```

Тестване с Coverage



Coverage: DatabaseTest X

100% classes, 78% lines covered in package 'p06_TirePressureMonitoringSystem'

Element	Class, %	Method, %	Line, %
Alarm	100% (1/1)	100% (3/3)	100% (9/9)
Sensor	100% (1/1)	33% (1/3)	40% (2/5)

```

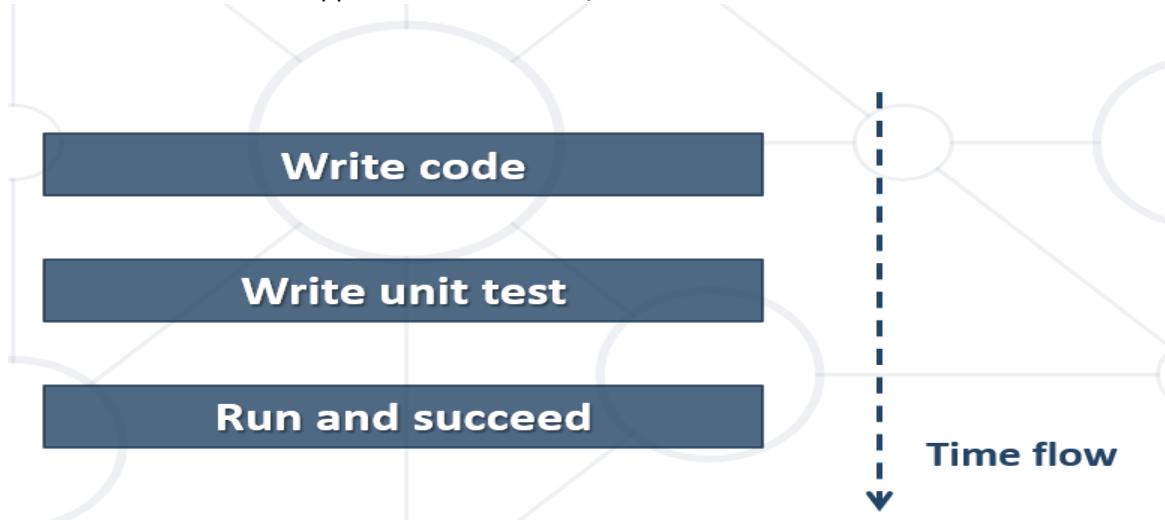
66     List<Person> people = ne;
67
68     if (username == null) {
69         throw new OperationInvali...
70     }
71
72     for (Person person : ele...
73         if (person == null)
74             continue;
75
76         if (person.getUsername()...
77             people.add(person);
78     }
79
80 }
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

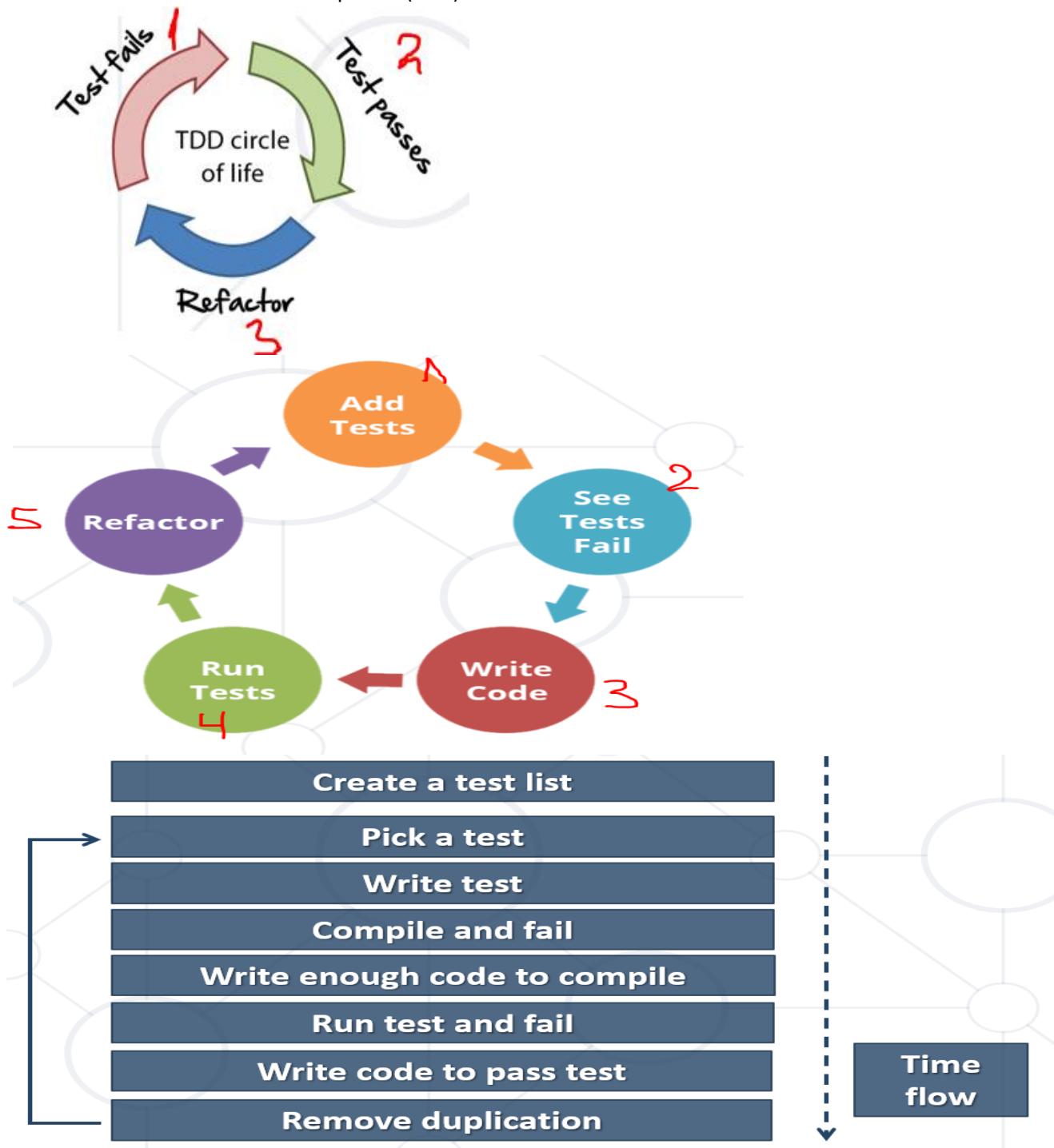
34. Test-Driven Development

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

35. Mutation testing

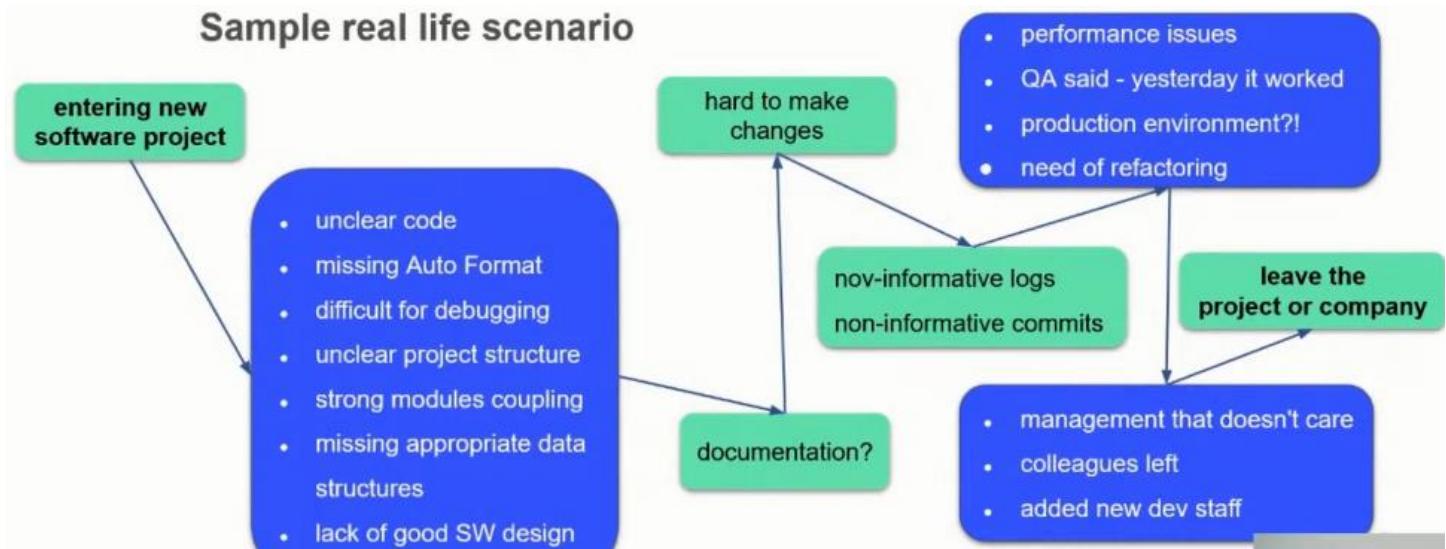
Mutation testing - Mutation testing, also known as code mutation testing, is a **form of white box testing** in which testers change specific components of an application's source code to ensure a software test suite can detect the changes.

Прави много на брой копия на компилирания байткод - като всяко копие е с някаква малка промяна - ако е `>`, то тества с `>=`, с `<` и гледа дали ако се смени някоя if проверка дали ще даде грешка. Има различни типове мутации (по категориите са). Ако правилно си написал кода, няма смисъл от тези mutation тестове.

Но има поне 2 причини да се използват - ако логиката е много сложна и просто програмиста пропусне някакъв edge case, или пък програмиста е писал излишен код - този вид тестове веднага ще го хванат!

36. Refactoring, good and bad coding practices

Sample real life scenario



Discovery of bad code

Manually by experienced devs.

Or via static analytics tools:

- Enforce naming, formatting and structural rules
- Apply code patterns to determine code smells
- Calculate **source code metrics** based on which to determine 'smelly' pieces of code

Source code metrics:

- Lines of code (LOC)
- Cyclomatic complexity (CC) - Показател колко условни конструкции и цикли има
- Coupling between objects (CBO)
- Lack of cohesion of methods (LCOM)
- Response for a class (RFC)
- Depth of inheritance tree (DIT)
- Number of parameters (NP)

List of tools for Java software metrics:

<https://www.monperrus.net/martin/java-metrics>

Many of the static analysis tools provide calculation of source code metrics in addition (such as SonarQube).

Some examples indicating the need of code improvements:

- LOC > 80
- CC > 10
- NP > 4
- DIT > 7
- These rules can be defined in a different way per project

Static analytics tools

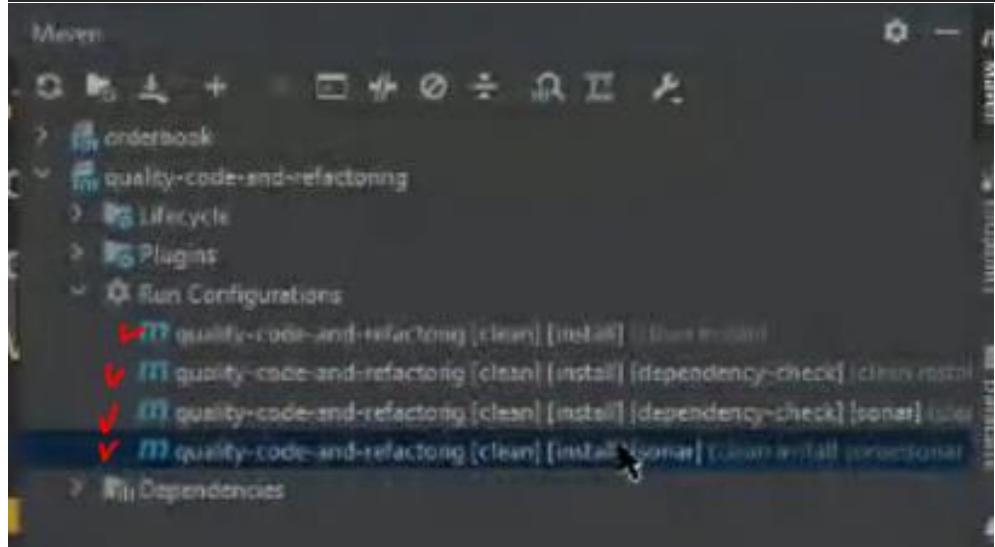
- **SonarQube** - пуска се през Docker

При всеки build да се извършва проверка от SonarQube. И ако има проблем, то да не преминава към следваща стъпка - на деплой примерно.

В случая локален Docker не е ок за работна среда. Обикновено използването на SonarQube се интегрира в Jenkins или GitHub actions CI/CD pipelines!!! И ако нещо не отговаря, то не се build-ва/не се деплоява.

Може да се пусне и в комбинация с OWASP Dependency Check.

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.sonarsource.scanner.maven</groupId>
        <artifactId>sonar-maven-plugin</artifactId>
        <version>${sonar-maven-plugin.version}</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```



- **SonarLint** - като плъгин в IntelliJ
- Checkstyle
- **PMD** - като плъгин в IntelliJ
- FindBugs / SpotBugs
- Facebook Infer
- DesigniteJava
- Google Error Prone
- Quilce (combines Checkstyle, PMD and Maven plug-ins)

Some tools provide specifically vulnerability scanning capabilities:

- Veracode
- **OWASP Dependency Check** (препоръки/практики свързани със сигурността - на самите библиотеки дали има пробив в тях)

С течение на времето, комбинация от библиотеки които са били без проблемни, то може да се окажат вече като vulnerable!

```
<dependency>
    <groupId>org.owasp</groupId>
    <artifactId>dependency-check-maven</artifactId>
    <version>8.4.0</version>
</dependency>
```

```
<plugin>
    <groupId>org.owasp</groupId>
    <artifactId>dependency-check-maven</artifactId>
    <version>8.4.0</version>
    <configuration>
        <format>ALL</format>
    </configuration>
```

- Snyk
- Eclipse CogniCrypt

Naming our objects to make it easier for everyone

- We must understand all about the object having in mind its name
- There should be chosen naming convention for the whole project
- To name our object only using English
- To use an optimal length - not short limited but also not too long names
- To avoid words abbreviations in most cases
- To avoid using numbers in the names
- Avoid misleading names
- Avoid identical sounding or difficult to pronounce / memorize names
- Use an analogy in naming if it is appropriate
- The names must be easy to search in the IDE
- The names must be created in such a way so that to be same as the IDE AutoComplete functionality

```
// Наименуване чрез комбинация от два езика
int broiSymbols = 80;
List<Company> allKontragentiData;
DebitSmetka dbtSmetka = new DebitSmetka();
TeacherDetails danniForTeacher;
```

```
// Наименуване чрез неуместни съкращения
LGEEOBNPRPNEFZL LGEEOBNPRPNEFZL;
String gentkn = "ABCD1234";
MasterAssAccess masterAss;
IMultiLanguageProperties props;
```

```
// Заблуждаващи имена
String currentFile = "/configuration/application.yaml";
```

```
// Дълго и трудно за запомняне име
Customer contact
    = getContactIfUserIsContactOfLoggedInUser("UID:555667");
```

```
// Наименуване само на английски език
int numberofSymbols = 80;
List<Company> companies;
DebitAccount debitAccount = new DebitAccount();
TeacherDetails teacherDetails;
```

```
// Наименуване без да използваме съкращения
CriminalPerson criminalPerson;
String generatedToken = "ABCD1234";
MasterAssistantAccess assistantAccess;
IMultiLanguageProperties bgProperties = new BGProperties();
```

```
// Съответстващи имена
String configurationPath = "../configuration/application.yaml";
```

```
// Логично и лесно за запомняне име
Customer customer
    = getCustomerById("UID:555667");
```



How to create 'quality classes'

- Class must represent real life objects
- Class objects must have the ability to be consistently created
- They must have well defined class actions
- It's better to have several classes instead of one Frankenstein class
- To have mutators (setters) only if it's needed and accessors (for example public only where needed)
- TODOs on non finished work - only when such policy in the company of soon completing the TODOs
- Class naming convention
 - Usually the name must be noun or with combination of adjective or verb
 - **Customer, EmailSender, DocumentBuilderFactory, TableColumnModelEvent**
- Interfaces naming convention
 - Usually use adjective or abstract noun
 - **Scrollable, Clickable, Vehicle, InputStream**
- Enumerations naming convention
 - The enumeration name is noun and the values are its types with upper letters
 - **DocumentType.MS_WORD, Color.RED**

The Optimal class length:

- It is defined according to class responsibility

- The classes should be as small as possible

Logical structure of the classes:

- Class fields should be clearly defined
- **Arrangement by convention** - the logger and constants first, then fields, then constructors, with appropriate access modifiers, mutators (setters), methods
- Instead of using overload constructors use descriptive FactoryMethods (the Factory Pattern)
- Avoid classes having unused methods

Classes interaction:

- There must be clearly defined relationships between classes
- There must be clearly defined classes hierarchy
- The implementation is hidden as much as possible from the outside world

How to create 'quality methods'

- Every method must do only one thing and do it very well
- There must not be an `super universal` method
- Method length must be as short as possible
- The methods must have the appropriate visibility
- Methods parameters must be well defined, think of validation
- Exceptions must be defined in a good way

Method naming strategies:

- Method name must be a **verb**, it is an **action** - `readOrderDetails()`
- The name must be descriptive enough, but not too long
- If the name is too long - there is a bad cohesion - прави повече неща понеже - екстрактваме/разбиваме метода на няколко метода тогава
- Avoid too abstract names - `result()`, `do()`, `make()`, `process()`
- Method names can be method **result descriptive** - `getCustomerIdentifier()`
- Use logically appropriate names/verbs - `read()` - `write()`, `open()` - `close()`

Method parameters:

- Use optimal number of parameters (max 7)
- Parameters must have well defined meaningful names
- Must be arranged by their importance
- To have similar arrangement for analogical methods
- If it's possible to combine some parameters in class data holders
- If it's necessary some methods parameters should be documented

```

// Лоша когезия на метод
validateDefinePersonIdentifierCheckRelationsAndSaveData

// Имащи значение имена на методи
validate();
defineIdentifier();
checkRelations();
save();

// Не носещи информация за абстрактни имена
make();
prepare("customerId:333");
load();

doTheMagic();

// Не добро извикване на не добре дефиниран метод
drawLine(3, 18, 123, 12, 4, new PaperDrawingBoard());

}

// Неправилно дефинирани параметри на метод
private void drawLine(int x1, int x2, int col,
                      int y1, int y2, DrawingBoard board) {
    // ...
}

// Методи, които вършат едно действие
validateAndSaveData();
prepareContracts("customerId:333");
loadUserDocuments("userId:23");

PersonDocument identityCard = new IdentityCard("userId:23");
Certificate userCertificate = currentUser.getCertificate();
signDocument(identityCard, userCertificate);

// Правилно извикване на по-добре дефиниран метод
Point pointA = new Point(3, 12);
Point pointB = new Point(18, 4);
DrawingBoard paperBoard = PaperDrawingBoard.getInstance();

drawLine(pointA, pointB, paperBoard, DrawingColor.BLUE);

}

// Правилно дефинирани параметри на метод
private void drawLine(Point pointA, Point pointB,
                      DrawingBoard board, DrawingColor color) {
    // ...
}

```

VS

Variables

- A variable must store only one data
- They must have right names according to their purpose and scope
- To have *minimal* life and appropriate scope - да се декларира непосредствено преди да се използва
- To be initialized on the right place
- A good approach is to store complex expressions result in variable

Variable naming rules:

- Their name must describe accurately stored data
- It is not necessary class fields to have in their names the class name
- Variable name depends on their scope
- Sometimes it is better to name according to its business value, not data structure
- Rather seen non informative abstract names- do not do that way!: temp, flag, value, i, result
- Naming of boolean variables - isDocumentLoaded, ordersCounter

Constant naming rules: - MILLISECONDS_IN_A_SECOND

```

// Неправилно именуване на методи
security.token(); // Правилно именуване на методи
security.generateToken();

// Неправилно именуване на полета на клас Person
String personFirstName;
String personLastName;
int personAge;
Address personAddress; // Правилно именуване на полета на клас Person
String firstName;
String lastName;
int age;
Address address;

// Не даващи информация имена на променливи
float value;
String temp;
boolean flag;
int i;
int exp;
boolean free; // Даващи информация имена на променливи
float shapeArea;
String generatedOTP;
boolean isUserLogged;
int currentDocumentIndex;
int tokenExpirationInSeconds;
boolean isPortAvailable;

// Използване на неаналогични имена за аналогични действия
FileReader confOpen;
FileWriter confStore; // Използване на аналогични имена според аналогични действия
FileReader configurationReader;
FileWriter configurationWriter;

// Пример с 'магическо число' в кода и други...
List<District> districts = new ArrayList<District>(28); // Пример с пълна яснота какво се случва в кода
final short DISTRICTS_IN_BULGARIA = 28; // трябва да бъде на ниво клас
List<District> districts = new ArrayList<District>(DISTRICTS_IN_BULGARIA);

for (int j = 0; j < 28; j++) {
    System.out.println("Община " + districts.get(j).getName()
        + " има " + districts.get(j).getCitizens()
        + " жители");
}

for (District district : districts) {
    String districtName = district.getName();
    int districtCitizens = district.getCitizens();

    // TODO тук да се използва StringBuilder или StringBuffer
    String districtDetails = "Община " + districtName + " има "
        + districtCitizens + " жители";

    System.out.println(districtDetails);
}

```

VS

Expressions

- The rule - one operation on a code line

This one is not ok

```
sideDifference++; a--; int b = sideA * sideB + surface; text = text + ", " + side
```

- Avoid chain constructions - нарушава се принципа **talk only to your friends** - ако имаме композиция, и може да се случи да зададем неправомерно дадена стойност/да инициализираме с по-малко на брой полета даден обект от изискуемите по проекта. Или да има енкапсулация - един обект да не може да бърка в друг обект. Иначе ако го вземем само като builder pattern това си е ок. `bmw.getEngine().prepareIgnition().startIt().setSpeed(50);`
- Common seen expressions -> method and result in variable with suitable name
- The result of complex expressions stored in informative variable
- Different expressions must be separated each other
- Create variable before returning in methods

```
if (executedAttempts == retryAttemptsAllowed && lastIterationExecutedSuccessfully == false)
    log.debug("All " + retryAttemptsAllowed + " allowed retry attempts for sending email were unsuccessful!");
```

VS

```
hasFailedForAllAttempts = (executedAttempts == retryAttemptsAllowed && lastIterationExecutedSuccessfully == false);

if (hasFailedForAllAttempts) {
    log.debug("All " + retryAttemptsAllowed + " allowed retry attempts for sending email were unsuccessful!");
}
```

```

return Jwts.builder()
    .setClaims(claims)
    .setSubject(username)
    .setIssuedAt(new Date())
    .setExpiration(new Date(new Date().getTime() + s * 1000))
    .compact();

```

```

Date tokenCreationDate = new Date();
Date tokenExpirationDate = new Date(tokenCreationDate.getTime()
    + (tokenExpirationInSeconds * MILLISECONDS_IN_A_SECOND));
String token = Jwts.builder()
    .setClaims(claims)
    .setSubject(username)
    .setIssuedAt(tokenCreationDate)
    .setExpiration(tokenExpirationDate)
    .compact();
return token;

```

VS

```

// Chain конструкция
Car bmw = BMW.newInstance("E60");
bmw.getEngine().prepareIgnition().startIt().setSpeedTo(50);

```

```

// Сложният израз е разбит на няколко последователни операции
Car car = CarFactory.createNew(BMWModels.E60);
int speed = 50;

Engine engine = car.getEngine();
Ignition ignition = engine.prepareIgnition();
engine = ignition.startIt();
car = engine.setSpeedTo(speed);

// Но по-правилно би било всеки клас да знае само за прекия си клас
// в методите на Сар да се работи само с обекта Engine,
// а обекта Engine да борави само с обекта Ignition и т.н.
car = CarFactory.createNew(BMWModels.E60);
speed = 50;

car.setSpeed(speed);

```

Conditional operations

- It's better to have **not complex** conditional statement and short body
- The condition must be easy to understand - **isUserSaved**
- If the condition is assigned to well named variable it is understandable
- The condition's block must always be in brackets
- For every case of the switch must be called a meaningful method
- Avoid deeply embedded conditional expressions

```

private static Enchantment getEnchant(String name) {
    if (!name.equalsIgnoreCase("sharpness") && !name.equalsIgnoreCase("afiada")) {
        if (!name.equalsIgnoreCase("baneofarthropods") && !name.equalsIgnoreCase("ruinadosartropodes")) {
            if (!name.equalsIgnoreCase("smite") && !name.equalsIgnoreCase("julgamento")) {
                if (!name.equalsIgnoreCase("efficiency") && !name.equalsIgnoreCase("eficiencia")) {
                    if (!name.equalsIgnoreCase("unbreaking") && !name.equalsIgnoreCase("inquebravel")) {
                        if (!name.equalsIgnoreCase("fireaspect") && !name.equalsIgnoreCase("aspectoflamejante")) {
                            if (!name.equalsIgnoreCase("knockback") && !name.equalsIgnoreCase("repulsao")) {
                                if (!name.equalsIgnoreCase("fortune") && !name.equalsIgnoreCase("fortuna")) {
                                    if (!name.equalsIgnoreCase("looting") && !name.equalsIgnoreCase("pilhagem")) {
                                        if (!name.equalsIgnoreCase("respiration") &&
                                            !name.equalsIgnoreCase("respiracao")) {
                                            if (!name.equalsIgnoreCase("protection") &&
                                                !name.equalsIgnoreCase("protecao")) {
                                                if (!name.equalsIgnoreCase("explosionsprotection") &&
                                                    !name.equalsIgnoreCase("protecaocontraexplosao")) {
                                                    if (!name.equalsIgnoreCase("featherfalling") &&
                                                        !name.equalsIgnoreCase("pesopena")) {
                                                        if (!name.equalsIgnoreCase("fireprotection") &&
                                                            !name.equalsIgnoreCase("protecaocontrafogo")) {
                                                            if (!name.equalsIgnoreCase("projectileprotection") &&
                                                                !name.equalsIgnoreCase(
                                                                "protecaocontraprojeteiis")) {
                                                                if (!name.equalsIgnoreCase("silktouch") &&
                                                                    !name.equalsIgnoreCase("toquesuave")) {

```

Cycles

- To choose optimal number of iterations
- The iteration helper variables to have meaningful names if needed
- Avoid unnecessary activities in the cycle body
- Avoid concatenate strings in the cycle body - конкатенацията на Стинг е по-тежка операция отколкото StringBuilder например

- Avoid usage of nested cycles - ако имаме дълбочина 3, значи трябва да променим нещо в логиката
- Have in mind collections changes during iterations

Exceptions

- They should contain only throwable operations in exception body - слагаме в try блока само операции, които могат да хвърлят exception. Иначе ако сложим всичко в try блока, то би се изпълнило по-бавно
- There must exist specific application exception for each layer - за всеки слой
- Exceptions must be processed according their probability to be raised - последователността им
- We must log all data when exception is raised to be easy for analysis - в catch блока да логваме аджаба
- After catch -> throw new exception corresponding to current layer - не е добре да се гълтат exception-и, а да се пропагират нагоре по слоевете.
- There must be no empty catch block, resources must be closed

```
String schemaLocation = AuthenticatorConfiguration.get_SAML_SchemaLocation();
File schemaFile = new File(schemaLocation);
Schema xmlSchema = null;

try {
    xmlSchema = schemaFactory.newSchema(schemaFile);
} catch (SAXException e) {
    String exceptionMessage = "Given SAML XML Schema is not valid, error creating schema by " + schemaLocation;
    throw new AuthenticatorException(exceptionMessage, e, MTITSExceptionType.SYSTEM_EXCEPTION);
}
```

Refactoring

What is refactoring and when it can be applied

- Actions for code improvement
- **The need of refactoring arises naturally**
- A process that can be performed periodically
 - While we write new code we usually get ideas how to improve the existing one
 - After analysis of existing code
 - After finish a project - we think how we can improve the code
 - Sometimes planned at certain stages

The need of refactoring - when?

- Changing the software design to provide improvements
- When a class does not have a strong cohesion
- There is a strong dependence on a particular class
- Divide a class into several classes or create a hierarchy of classes
- When a change requires changes elsewhere
- When general functionality can be extracted to a base class
- If it is clear that applying a Design Pattern will lead to improvements
- To transform the logic to depend to abstraction and not to implementation
- Common seen/repeated operations / expressions -> extract to new method
- To obtain used resources optimization
- To establish a new class containing related data fields - например x1, y1, x2, y2 да бъде изнесено в обект Point(x, y)
- When reducing the class fields visibility
- A long method can be divided into several methods
- Extract a piece of code into a new method
- When a method has a long list of parameters
- The complex expressions to be 'simplified' in a new method - сложен израз може да се изнесе направо в метод
- When moving methods to another class or level of abstraction

- When improving names of classes, methods and variables
- Introducing of new variables for more clarifying code
- Substituting ‘magic numbers’ in the code with meaningful variables/constants
- Introducing constants or enumerations on local or global level
- Eliminate deep nested cycling blocks
- Simplification of conditions in condition statements

Refactoring features / характерни черти

- It is better to start well designed code from the beginning
- To be careful not to break working code
- To be sure that after the refactoring, the code will work the same way
- To be sure that tests pass successfully again
- To be done on stages and be able to bring back to previous state
- Usually in time we will get new ideas for code quality improvements
- Interfaces are needed only when there will be different implementations

Categories of code smells

Code smells can be categorized by their level of appearance:

- application
- class
- method

I. Application level code smells

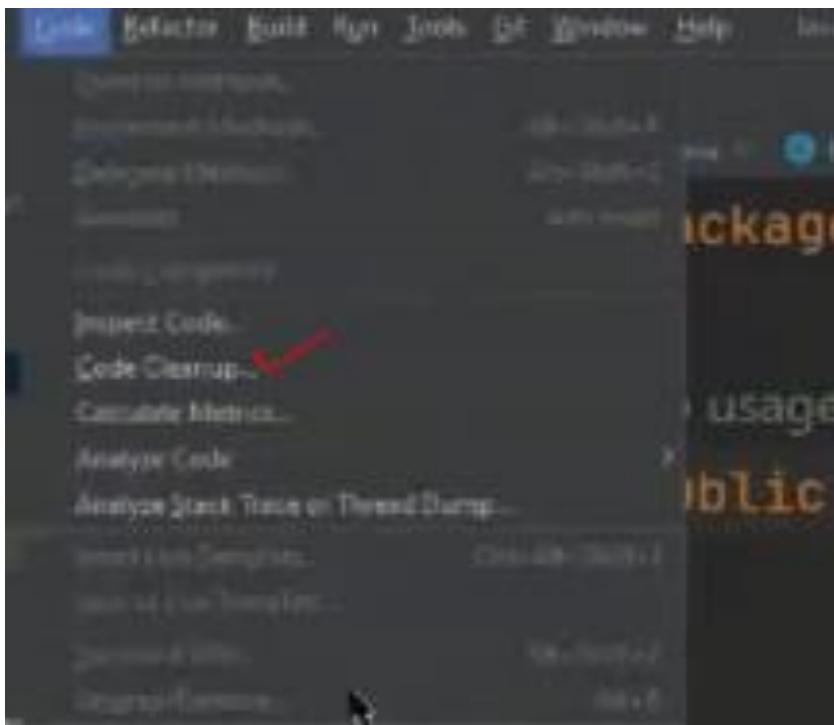
unused /dead code

- Certain types of unused (dead) code such as unused local variables or private fields/methods can be detected by the IDE and by static analytics tools.
- Other forms of dead code such as unused classes or unused public methods can be identified on the basis of manual code review

Resolution:

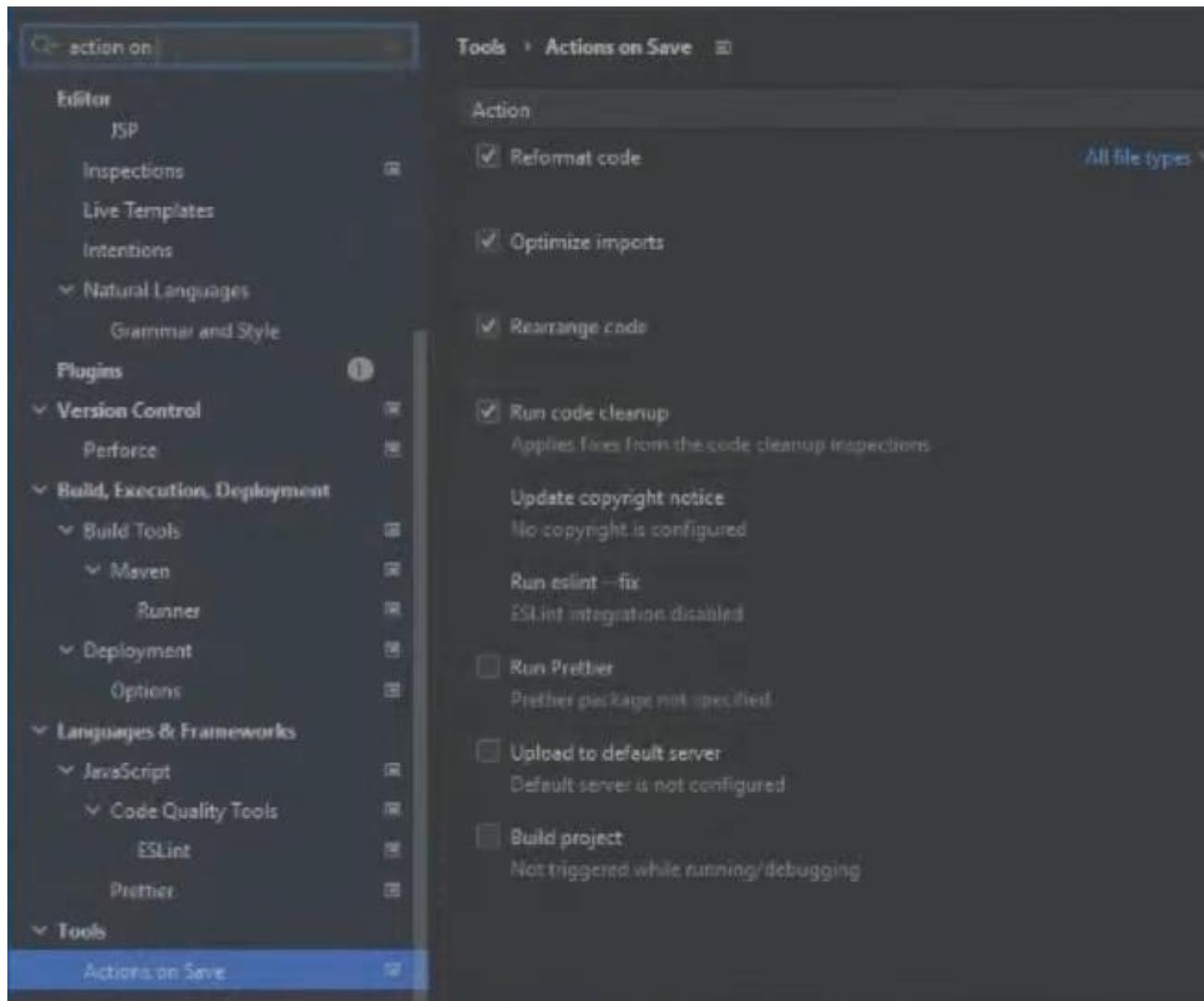
- IDEs such as IntelliJ IDEA provide a ‘code cleanup’ feature that allows for automatic removal of identified dead code
- Otherwise manual removal of entire files or blocks of code is to be done
- In any case removal of dead code improves maintainability of the system

Code cleanup in IntelliJ



Actions on Save (след експлицитен Save от наша страна и като има промяна все пак де).

Но по-добре **изрично и ръчно** да цъкнем Ctrl+Alt+O за импортите и Ctrl + Alt + L за автоматично форматиране - защото като цъкнем ръчно ще отидем да видим дали са се махнали импортите, а при Save няма да отидем да проверим. Въпрос на стил на работа.



duplicated code

- Duplicated code can be either intentional or unintentional
- Developers tend to intentionally copy-paste existing code and modify slightly instead of introducing a common abstraction
- As the system evolves, some leftover code remains unintentionally undeleted

Resolution:

- Create a common abstraction or utility that eliminates duplicate code
- In case of duplicate libraries, try to find a single version to use in all cases
- Static analysis tools and IDEs assist in discovery of duplicated code
- If we refer to project dependencies we can also use:
 - **duplicate-finder-maven-plugin** to find duplicate classes on the classpath

```

<plugin>
  <groupId>com.ning.maven.plugins</groupId>
  <artifactId>duplicate-finder-maven-plugin</artifactId>
  <executions>
    <execution>
      <phase>verify</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

- **maven-enforcer-plugin** to check for duplicate libraries on the classpath
- use the **dependency:tree** goal of the **maven-dependency-plugin** to manually inspect the dependency tree of a project and identify duplicate libraries.

[large and unstructured project](#)

- A project contains a large number of classes, many of which are big and complex
- There are many big and complex methods in the project
- Project provides logic for different business domains
- Често разработчика, особено ако е нов или не е запознат напълно с големия проект, прави/имплементира логика, която вече я има в друг подпроект и може да се използва наготово.

Resolution:

- The project may need to be split either into smaller subprojects
- Or the classes and the methods in the project needs to be split into smaller classes and methods

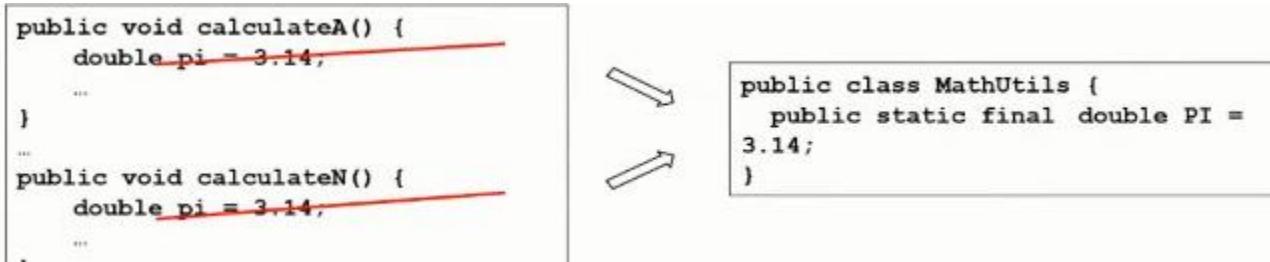
[spaghetti/lasagna/ravioli code](#)

In any of these cases typically general refactoring needs to be applied in the project:

- Merge some of the excessive layers in the system in case of lasagna code
- Refactor or eliminate unstructured logic in the project in case of spaghetti code - омешани единия клас в другия клас и код без принципите на ООП
- Merge some of loosely coupled components to larger ones based on logical grouping in case of ravioli code

[shotgun surgery](#)

Shotgun surgery occurs when **a single change** needs **to be applied to multiple classes/places** at the same time



[input kludge](#)

- Input kludge is the lack of validation of user input
- It may cause unexpected exceptions, crash the system or open security vulnerabilities

```

public static void main (String[] args) {

    String query = args[1];
    ...
    // is it a valid SQL query ?
    statement.executeQuery(query);
}

```

Resolution:

- Provide proper input validation along with a proper mechanism to convey validation errors to the user

```

public static void main (String[] args) {

    String query = args[1];
    validateQuery(query);
    ...
    statement.executeQuery(query);
}

```

hardcoding

- Hardcoding happens when we embed data such as integer constants or text directly inside the source code - да не пишем в кода например числото 28 като номер на община, а да бъде в отделна наименована константа
- Every time a hardcoded value needs to be changed, it requires recompilation

Resolution:

- Store hardcoded values externally and reference them from the application
- These external sources might be a properties file, a normal Java file for constants, RDBMS and so on
- Requires additional logic for reading of values in the application but improves maintainability

softcoding

- Softcoding is the opposite of hardcoding
- It is typically the act of storing externally more values than needed even ones that won't/cannot be changed or do not target the audience of users - например всички части от url-а ако за изнесени в отделно място, а то има нужда само една от всичките части на url-а да бъде променяна изобщо някога
- Makes the system difficult to configure properly

Resolution:

- Eliminate some of the complex configuration and values that are not going to change

excessive calls to third-party systems

- These could be executing queries against the RDBMS, web service calls, Elasticsearch calls, etc.
- Having too many calls to external systems makes the system difficult to maintain and also increases latency

Resolution:

- Remove some of the calls to third party systems if possible with application logic
- Batch calls to third party systems if that is provided as a capability (for example ElasticSearch and JDBC provides batching of queries)
- Merge multiple calls if possible (for example if you execute multiple SELECT queries on related data against the RDBMS you can use JOIN instead and merge them into a single query)
- Change the external system and change the code in our app so that we make less or only 1 http call instead of 10 calls for example.

usage of vendor code

- Directly copying-pasting vendor code (whether proprietary or open source) and modifying it may incur license violations
- In addition application developers become responsible for maintaining the vendor code being copied - копират готов код, но този готов код прави повече отколкото е нужно за нашия случай

Resolution:

- Remove vendor code and introduce a proper library if possible
- If no proper library available or difficult to extend the logic provided, then use an alternative library or write your own logic

defensive programming

- Defensive programming refers to a practice where developers tend to be "overly" protective
- Examples include excessive input validation, unnecessary checks for null, etc.

```
// product ID can never be null
if(product.getId() != null) {
    validateProduct(product);
}
```

За по стари системи директно assert a > b;

lack of proper comments

- Many practitioners promote the fact that the course code needs to be "self-documented" - т.е. да няма никъде коментари и сами от кода да разбираме какво прави кода
- While this is true, in many cases there are certain situations where comments are needed. These include for example methods that implement certain algorithms or classes that provide complex business logic

meaningless/misleading comments

Не носят повече информация:

```
// assigning product count
int productCount = products.size();

// retrieve records from MySQL
Records records = mongoDbUtil.getRecords()
```

boat anchor

- Boat anchor refers to a piece of code that serves no particular purpose in the current project
- It is typically source code that has been intentionally added for (eventual) future use
- Similar to dead/unused code antipattern where code has been in many cases either used in previous version or unintentionally added
- Но не се е наложило да бъде използван този код

```
public Configuration initConfiguration() {
    Configuration config = readConfiguration();
    writeConfigurationToDB(config);
    return config;
}
```

writeConfigurationToDB() writes configuration to the RDBMS but it is not used by the application (or other applications)

golden hammer

- Golden hammer is a software technology or concept applied obsessively in the project and based on previous usage

Example: I made the system to use the NoSQL database **NoOneIsReallyUsingThatMuch** and its API because it is so cool and I used it in two of my previous projects.

dead end

- A dead end refers to a library or component that is modified by developers but is no longer maintained and supported by the supplier
- In that manner the support burden transfers to the application developers

Example: We use a patched version of the **ESAPI** library from 2011 for input validation that is no longer supported but there are several critical security issues uncovered.

II. Class level code smells

large (god) class

- Large (aka god) class may refer to classes that are too big in terms of lines of code or methods provided
- In a slightly different manner a god class may refer to a class that is highly complex (also referred to as "brain class")

Resolution:

- In most cases the standard practice is to extract extra classes from the god class
- In certain situations it is also sufficient to extract methods from the god class to existing classes

lack of cohesion

- **Cohesion refers to the degree at which the components of the class relate to each other**
- If a class provides multiple distinct roles it has low cohesion/ сплотеност

```
public class UsersAndRolesManager {  
    ...  
}
```

- Some cases (such as above) might be more obvious based on the naming being used

Resolution:

- Distinct roles provided by the class need to be extracted to multiple other classes - например class UsersManager и class RolesManager
- Similar to god class in certain situations moving methods to existing classes also alleviates the lack of cohesion

feature envy

- Feature envy refers to a situation where the class uses more methods and fields from other classes than its own - клас, който разбира повече за друг клас отколкото за себе си
- A basic example is delegating extensively to setters and getters from other classes:

```
public class ValueHolder {  
    private ValueDTO valueDTO;  
  
    public String getName() {  
        return valueDTO.getName();  
    }  
    ...  
}
```

Resolution:

- In cases of simple delegation remove methods and use the referenced class
- In more complex scenarios moving methods and fields to the referenced class is a possible solution
- Extracting methods to the referenced class is also a possibility

inappropriate intimacy

- Similar to feature envy, but typically both classes are referring to each other and are typically used together
- From a slightly different aspect, one class can refer extensively to internal members of another class

```
public class User {  
    String name;  
  
    public UserDetails createDetails  
        (String name, String email) {  
        this.name = name;  
        return new UserDetails(mail);  
    }  
    ...  
}
```

```
public class UserDetails {  
  
    private User user;  
    public String getName() {  
        return user.name;  
    }  
    ...  
}
```

Resolution:

- Move the logic from one of the classes to the other by extracting methods and fields
- In the case of bidirectional communication if possible remove the relation from one of the classes to the other - да няма двустранна зависимост
- Колкото повече класове дефинираме, толкова повече classloading и повече памет се заема в приложението

excessive coupling

- Excessive coupling/обвързаност refers to the dependency of a class to many other classes:

Too much composition also here.

```
public class UserManager {  
    private EntityManager entityManager;  
  
    private AuthManager authManager;  
  
    private UserValidator validator;  
  
    ...  
}
```

- Coupling introduces **difficulty in extending and testing** the target class

Resolution:

- Coupling can be reduced by:
 - Introducing interfaces rather than concrete dependencies
 - Extracting a class
- Dependency injection frameworks help in reducing coupling

refused bequest / отказано наследство

- Refers to the scenario where a child class is not using ("refusing to use") logic from the parent class
- In certain scenarios the parent and child classes are not relating logically to each other - child класа user се отказва да използва функционалността идваща от UserUtils и прави нова дублираща логика.

```
public class UserUtils {  
    public boolean hasRole  
        (String role) {  
        ...  
    }  
    public String[]  
        getBlockedUsers() {  
        ...  
    }  
}
```

```
Public class User  
    extends UserUtils {  
  
    public void export () {  
        if(hasRole(  
            Roles.EXPORT)) {  
        ...  
    }  
}
```

Resolution:

- Introduce composition & delegation rather than sub-classing in related classes or rather than inheriting
- Extract methods from the parent class - в parent класа да има повече разделение на методите кой какво прави
- Extract interfaces from parent class and make interested children inherit from them

lazy class / freeloader

- A class that does too little and is used rarely
- Lazy classes might be:
 - Classes introduced with the intention to be used in future
 - Classes that have reduced use over time

Resolution:

- Lazy classes may be removed - а съдържанието (ако е ползва все пак) на класа да се измести в друг клас - там където ще се използва

indecent exposure

- Indecent exposure / неподходящо разкриване occurs when a class exposes more of its internal structure to clients than needed
- These are typically fields and methods that need to be private or at least having package/protected access but are marked as public
- В някои компании обаче можем да срещнем полета и/или всичко да е public

downcasting

constant class

data clump

- A data clump refers to a set of classes typically used together

```
public class AuthManager {  
  
    private User user;  
  
    private UserDetails userDetails;  
  
    ...  
}
```

Resolution:

- Extract methods from to one of the classes in the data clump
- Introduce a wrapper object (for example from User and UserDetails) that encapsulates the data clump

poltergeist class

- A short lived, typically stateless class used to provide initialization or supports the operations of other classes

Например клас Initializer, който се използва само веднъж при стартирането на приложението, и повече никога не се използва. Не е нужно да създаваме такива класове. Може логиката при инициализация да бъде като метод анотиран с @PostConstruct в друг клас правещ повече неща.

sequential coupling

- Sequential coupling refers to a situation where the methods of a particular class need to be used in a particular order by calling classes - Не е нужно и работа на класа Client да извиква всяко едно поотделно

```
public class Server {  
  
    public void initLogging() {  
        ...  
    }  
  
    public void initDB() {  
        ...  
    }  
}
```

```
public class Client {  
  
    private Server server;  
  
    public void createServer() {  
        server.initLogging();  
        server.initDB();  
    }  
}
```

Resolution:

- Introduce a facade method that implements the coupling sequence and hide details from callers

```
public class Server {  
  
    public void initialize() {  
        initLogging();  
        initDB();  
    }  
    ...  
}
```

```
public class Client {  
  
    private Server server;  
  
    public void createServer() {  
        server.initialize();  
    }  
}
```

large and complex hierarchies

III. Method level code smells

too many parameters

large cyclomatic complexity

Показател колко условни конструкции и цикли има в един метод - добре е да не е повече от 7

deep nesting

lack of cohesion

long method

excessively long/short identifiers

use of string concatenation

[use polymorphism instead of switch](#)

We can also use Map instead of switch. The map plays the role of polymorphism.

[primitive obsession](#)

[excessive return of data](#)

[excessively long line of code](#)

[busy waiting](#)

Да не влезем в безкраен цикъл - може да сложим try-catch

[error hiding](#)

Добре е поне да логнем грешката. Може и да пропагейтнем нагоре по слоевете по-генерален или custom-изиран Exception.

[magic numbers/strings](#)

Числови/стрингови константи да не ги хакаме директно, а с деклариране на константа

[comparing objects with ==](#)

[not checking for null](#)

[long message chains](#)

[resource leaks](#)

[Conclusion](#)

If we apply these principles, we will not refactor (often)

Making software design by using the power of OOP

- Using Abstraction and Polymorphism
- Loose coupling, String cohesion
- Prefer Composition instead of Inheritance
- Use appropriate **Design Patterns** where it is suitable
- Follow the principles of simplicity and reusability
- If it's designed with more classes -> the maintenance is easier

[Other good practices](#)

- To have a good informative logging mechanism
- Follow source control convention
- Static/dynamic code analysis instruments can be used
- It is good all the team members to be familiar what's going on
- Code review helps and improves code quality
- Make project retrospections on appropriate stages

37. Design Patterns

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

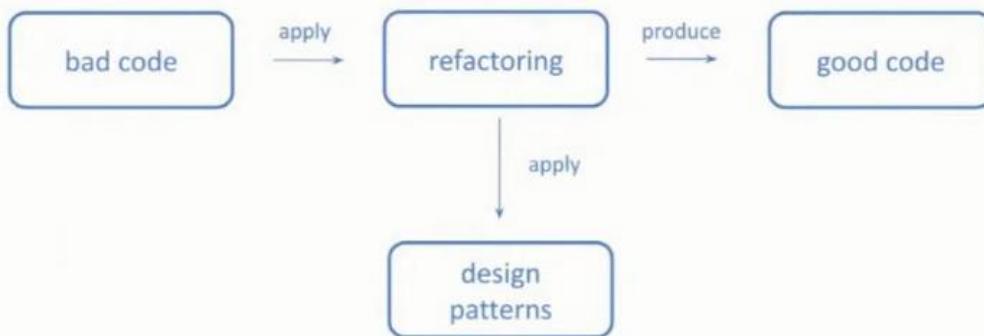
Drawbacks

- 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**

2. The big picture

Design patterns should be applied only when needed!

Design pattern is itself an abstraction.



Design patterns are well-known solutions to common problems in software development.

According to the fundamental design patterns book by "**Gang of Four**" and acc. to other resources, they are divided into three logical groups:

- **creational** - related to the creation of objects

- **structural** - related to the structural relationship between classes
- **behavioral** - related to the communication between objects

<https://github.com/martinfmi/Java-design-patterns>

<https://refactoring.guru/design-patterns/catalog>

3. 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

Abstract factory pattern

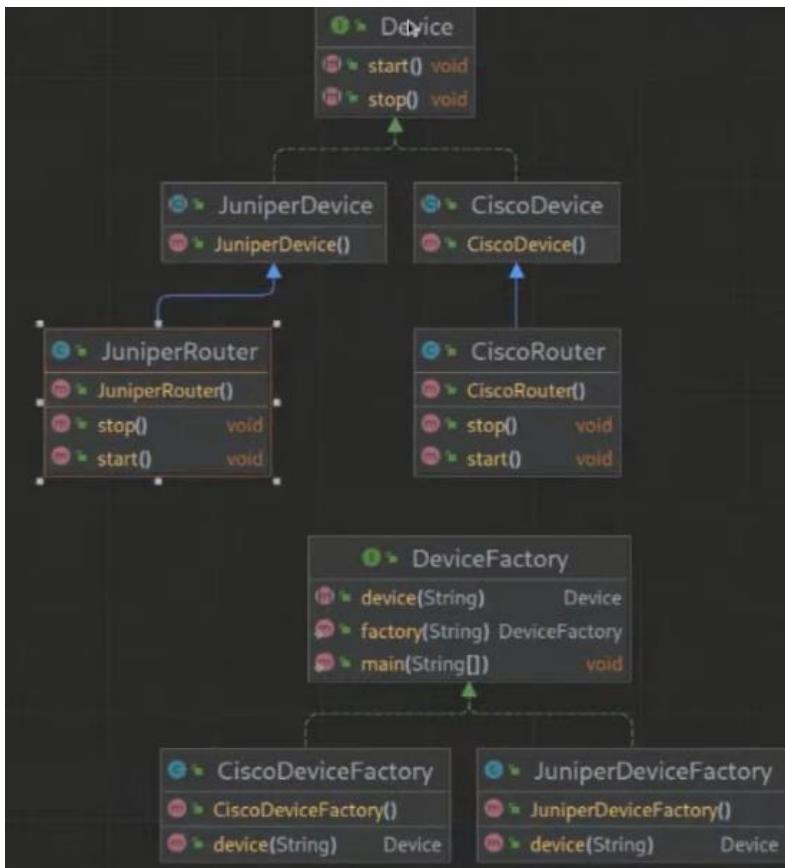
A factory is an OOP concept that denotes/обозначава an object to create another object. - **някакъв обект се използва за създаването на други обекти!** В малко по-широк смисъл хората наричат factory **всичко, което създава обекти и не е конструктор!**

Abstract factory provides an interface for creating a variety of dependent objects **without using their concrete classes.**

A good way to design functionality that:

- **Hides of how objects are created**
- **Provides independence between client classes and created objects**
- Provides a common mechanism to create instances of the group of dependent objects

Използва се примерно за мултиплатформено приложение.



```

public interface DeviceFactory {

    public Device device(String serialNumber);

    public static DeviceFactory factory(String vendor) {
        DeviceFactory factory = null;
        switch (vendor) {
        case "cisco":
            factory = new CiscoDeviceFactory();
            break;
        case "juniper":
            factory = new JuniperDeviceFactory();
            break;
        default:
            throw new RuntimeException(String.format("Unsupported vendor: %s", vendor));
        }
        return factory;
    }
}

public static void main(String[] args) {
    Device device = DeviceFactory.factory("cisco").device("router SN123");
}
}

```

```

public class CiscoDeviceFactory implements DeviceFactory {

    @Override
    public Device device(String serialNumber) {
        Device device = null;
        if(serialNumber != null && serialNumber.contains("router")) {
            device = new CiscoRouter();
        }
        return device;
    }
}

```

```

    } else {
        throw new RuntimeException(String.format("Unsupported device: %s", serialNumber));
    }
    return device;
}
}

-----
public class JuniperDeviceFactory implements DeviceFactory {

    @Override
    public Device device(String serialNumber) {
        Device device = null;
        if(serialNumber != null && serialNumber.contains("router")) {
            device = new JuniperRouter();
        } else {
            throw new RuntimeException(String.format("Unsupported device: %s", serialNumber));
        }
        return device;
    }
}

```

Factory method pattern

- Allows a class to create objects without knowing concrete implementation - когато искаме да скрием част от имплементацията
- Achieved by means of calling proper factory method **on a child object** that is responsible to create concrete implementation instance
- Например имаме 1 клас с няколко подкласа и искаме хем да извлечем някакви обобщаващи данни в класа, хем да имаме някаква гъвкавост
- Not to be confused with static factory methods that provide simpler mechanism to create objects instead of using a constructor with parameters
 - Consider static factory methods instead of constructors

Конструкторите в Джава обвързват конкретния конструктор с конкретния клас!

```

public abstract class DeviceController {

    public void start() {
        Device device = createDevice();
        // ... do something with device
    }

    public abstract Device createDevice();

    public static void main(String[] args) {
        new CiscoRouterController().start();
    }
}

-----
public class CiscoRouterController extends DeviceController {

    @Override
    public Device createDevice() {
        return new CiscoRouter();
    }
}

-----
public class JuniperRouterController extends DeviceController {

```

```

@Override
public Device createDevice() {
    return new JuniperRouter();
}
}

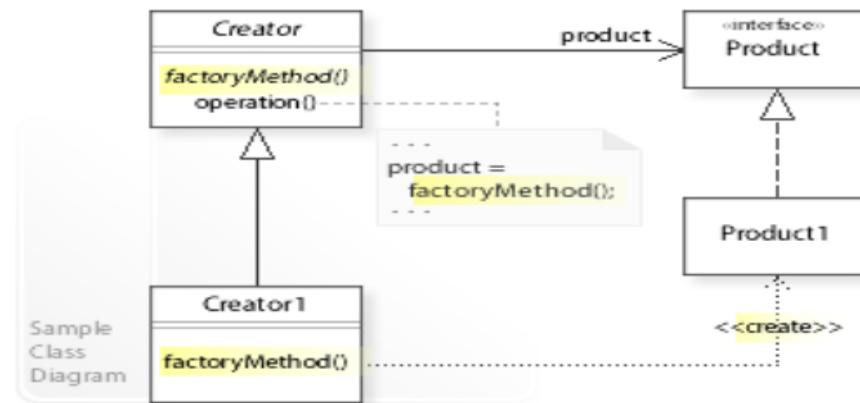
```

Factory methods in practice:

- JDK - javax.naming.spi.ObjectFactory (getObjectInstance() methods)
- Spring - org.springframework.beans.factory.BeanFactory (getBean() methods)

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("Miauo");
    }
}
public class Giraffe implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Муууу ааа");
    }
}
public class Monkey implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Уаааа-хахаха");
    }
}

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);
    }
}
```

Builder pattern

- Builder pattern is used to separate creation of complex objects from their representation.
- In practice many frameworks and libraries implement builder pattern with an internal Builder class.
- It is perfectly legal and accepted if the builder class is a top-level public class in its own file. It may also be hidden the builder....
 - Consider builder when having too many constructor parameters

```
public class Device {  
  
    private String serialNumber;  
    private String shortName;  
    private double price;  
  
    public Device(String serialNumber, String shortName, double price) {  
        this.serialNumber = serialNumber;  
        this.shortName = shortName;  
        this.price = price;  
    }  
  
    public String getSerialNumber() {  
        return serialNumber;  
    }  
  
    public String getShortName() {  
        return shortName;  
    }  
  
    public double getPrice() {  
        return price;  
    }  
  
    public static class Builder {  
  
        private String serialNumber;  
        private String shortName;  
        private double price;  
  
        public Builder serialNumber(String serialNumber) {  
            this.serialNumber = serialNumber;  
            return this;  
        }  
  
        public Builder shortName(String shortName) {  
            this.shortName = shortName;  
            return this;  
        }  
  
        public Builder price(double price) {  
            this.price = price;  
            return this;  
        }  
  
        public Device build() {  
            return new Device(serialNumber, shortName, price);  
        }  
    }  
  
    public static void main(String[] args) {  
        new Device.Builder().serialNumber("SN123").shortName("router").price(1000d).build();  
    }  
}
```

Builder in practice:

- java.util.stream.Stream.Builder
- java.util.Calendar.Builder
- java.util.Locale.Builder
- When you create setters automatically in IntelliJ with the builder pattern

Builder Pattern (Cascade method pattern) explained

Стъпка по стъпка създава обекта като имаме контрол на всяка стъпка.

Не създаваме конструктори с различен брой параметри, а правим chain-ване на сетьри.

- 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.address = address;  
        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();
    }
}

```

Dependency Injection

- Dependency injection is a way to apply **inversion of control**
- It happens when one object supplies the dependencies of another object
- Typically achieved with the support of a DI framework such as Spring DI or JavaEE CDI(Context And Dependency Injection)
- It provides a fundamental mechanism to loose coupling between objects - Искаме да бъдем гъвкави, постигане на loose coupling / намаляне на обвързаността между обектите
- Also provides a way for applications to support different configurations
 - Prefer dependency injection to hardwiring resources

An example without using annotations:

```

public class DeviceInjector {

    public void inject(DeviceController controller, Device device) {
        controller.setDevice(device);
    }

    public static void main(String[] args) {
        DeviceInjector injector = new DeviceInjector();
        Device router = new CiscoRouter();
        DeviceController controller = new DeviceController();
        injector.inject(controller, router);
    }
}

-----
public class DeviceController {

    private Device device;

    public void setDevice(Device device) {
        this.device = device;
    }
}

```

Lazy initialization

- It is a pattern used to delegate the creation of an object for a later time - i.e. created/initialized only when called

- Creation of the object happens typically when it is needed
- In many cases lazy initialization is combined with the factory method pattern
 - Use lazy initialization judiciously / разумно

В Java всеки обикновен клас се инициализира на практика lazy - т.е. когато се извика за първи път с думичката new!

```
public class CiscoRouterController extends DeviceController {
    private Device device;

    @Override
    public Device createDevice() {
        if(device == null) {
            device = new CiscoRouter();
        }

        return device;
    }
}
```

Example with the so called double locking:

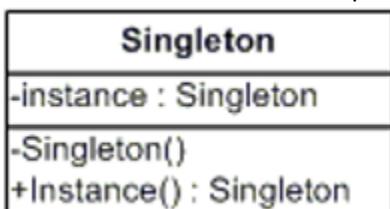
```
public class CiscoRouterSynchronizedController extends DeviceController {
    private volatile Device device;

    @Override
    public Device createDevice() {
        if(device == null) {
            synchronized (this) {
                if(device == null) {
                    device = new CiscoRouter();
                }
            }
        }

        return device;
    }
}
```

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"));
    }
}

```

Singleton – in more details

- Provides a mechanism to ensure a class has only one instance
- Used to avoid excessive creation of class instances whenever possible
- A more flexible alternative to static utilities
- A generalization of the singleton pattern is called ‘**multiton**’

- A multiton provides creation of multiple instances:
 - Enforces the singleton property with a private constructor or an enum type
 - Avoid creating unnecessary fields

Имплементация на Singleton design pattern - lazy and eager

Единствено енъм класа се инициализира при билдване на проекта, всички други сингълтъни се инициализират при Run-time.

1. In **eager initialization**, the instance of the singleton class is **created at the time of class loading**. The drawback to eager initialization is that the method is created even though the client application might not be using it.

Singleton could **also** be initialized **with static block**.

Both eager initialization and static block initialization create the instance even before it's being used and that is not the best practice to use!

```
public class EagerInitializedSingleton {

    private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();

    // private constructor to avoid client applications using the constructor
    private EagerInitializedSingleton() {}

    public static EagerInitializedSingleton getInstance() {
        return instance;
    }
}

-----
public class StaticBlockSingleton {
    private static final StaticBlockSingleton instance;
    private StaticBlockSingleton() {}

    // static block initialization for exception handling
    static {
        try {
            instance = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new RuntimeException("Exception occurred in creating singleton instance");
        }
    }

    public static StaticBlockSingleton getInstance() {
        return instance;
    }
}
```

2. Lazy initialization method to implement the singleton pattern creates the instance in the global access method.

```
public class LazyInitializedSingleton {

    private static final LazyInitializedSingleton instance;

    private LazyInitializedSingleton() {}

    public static LazyInitializedSingleton getInstance() {
        if (instance == null) {
```

```

        instance = new LazyInitializedSingleton();
    }
    return instance;
}
}

```

3. When using **static class** InstanceHolder, it initializes lazy!

```

public class LazyCiscoRouterControllerWithStatic {

//this class is not created/loaded until it is invoked!!! That's why it is lazy
// it is also thread-safe as class loading in java by the JVM is a thread-safe operation by
default!
private static class InstanceHolder {
    private static final LazyCiscoRouterControllerWithStatic controller = new
LazyCiscoRouterControllerWithStatic();
}

private LazyCiscoRouterControllerWithStatic() {
    System.out.println("Creating an instance");
}

public static LazyCiscoRouterControllerWithStatic instance() {
    return InstanceHolder.controller;
}

public static void main(String[] args) {
    System.out.println("Hello");

    // When we call for the 1st time it initializes
    System.out.println(instance());
    System.out.println(instance());
}
}

```

Result is:

```

Hello
Creating an instance
com.toshev.martin.patterns.creational.singleton.LazyCiscoRouterControllerWithStatic@4617c264
com.toshev.martin.patterns.creational.singleton.LazyCiscoRouterControllerWithStatic@4617c264

```

4. Thread Safe Singleton lazy - A simple way to create a thread-safe singleton class is to make the global access method synchronized so that only one thread can execute this method at a time.

```

public class ThreadSafeSingleton {
    private static final ThreadSafeSingleton instance;

    private ThreadSafeSingleton(){}

    public static synchronized ThreadSafeSingleton getInstance() {
        if (instance == null) {
            instance = new ThreadSafeSingleton();
        }

        return instance;
    }
}

```

5. Thread Safe **Double Locking** Singleton lazy

```

public class LazyCiscoRouterSynchronizedController {

    private static final volatile LazyCiscoRouterSynchronizedController controller;

    private LazyCiscoRouterSynchronizedController() {}
}

```

```

public static LazyCiscoRouterSynchronizedController instance() {
    if(controller == null) {
        synchronized (LazyCiscoRouterSynchronizedController.class) {
            if(controller == null) {
                controller = new LazyCiscoRouterSynchronizedController();
            }
        }
    }
    return controller;
}

```

6. Enum Singleton

Java ensures that any enum value is instantiated only once in a Java program. Since Java Enum values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible (for example, it does not allow lazy initialization) - i.e. **it is always eager**.

```

public enum EnumSingleton {
    INSTANCE;

    public static void doSomething() {
        // do something
    }
}

```

Object pool

- An object pool alleviates (облекчава) the need to create expensive objects
- Well known applications of the pattern are connection and thread pools
- **Когато пазим група обекти с цел да ги кешираме** - за да не инициализираме наново Connection или пък да не инициализираме нова нишка - тъй като това са скъпи процеси
- Ако хеша нарасне прекалено много, тогава Внимание - 1.garbage collector-а ще изtrieе ли нещо автоматично; 2. Ако не, то никаква стратегия за премахване на стари обекти. Сложно става с други думи.

```

public class CiscoDevicePool {
    private Map<String, Device> devicePool = new HashMap<String, Device>();

    public Device getDevice(String serialNumber) {
        Device device = devicePool.get(serialNumber);
        if (device == null) {
            device = new CiscoRouter();
            // set proper device settings ...
            devicePool.put(serialNumber, device);
        }

        return device;
    }
}

```

Object pool in practice are:

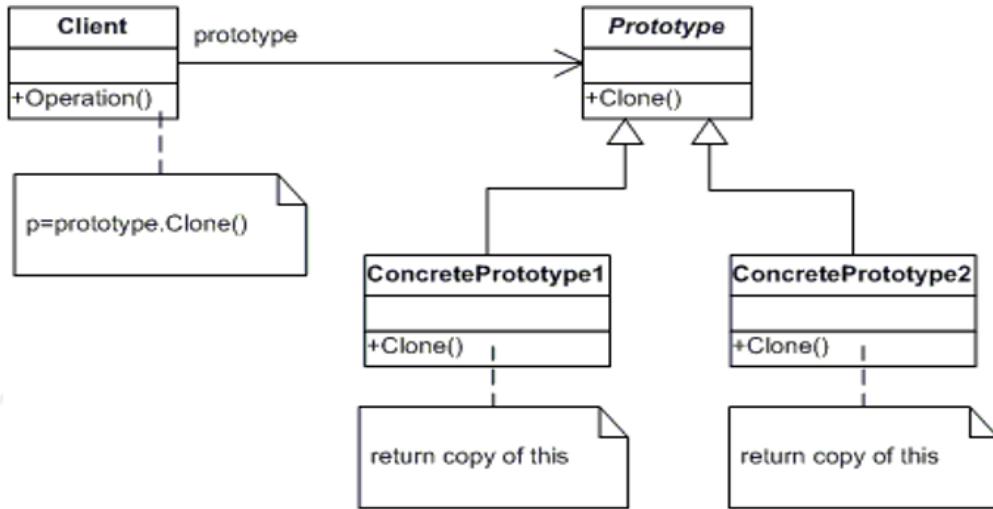
- JDBC connection pool
- JDK executor thread pools

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 of 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);
    }
}
  
```

- Provides a mechanism to create objects from a template object - от базов обект
- Used typically to avoid creation of expensive objects using 'new'
- Provides the ability to copy objects without knowing the concrete subtype
 - Override clone judiciously / изрично

```

public abstract class Device implements Cloneable {

    @Override
    protected abstract Device clone() throws CloneNotSupportedException;
}
  
```

```

-----  

public class CiscoDevice extends Device {

    private String serialNumber;
    private String shortName;
    private double price;
  
```

```

public CiscoDevice(String serialNumber, String shortName, double price) {
    this.serialNumber = serialNumber;
    this.shortName = shortName;
    this.price = price;
}

public String getSerialNumber() {
    return serialNumber;
}

public String getShortName() {
    return shortName;
}

public double getPrice() {
    return price;
}

@Override
protected Device clone() throws CloneNotSupportedException {
    return new CiscoDevice(serialNumber, shortName, price);
}
}

```

Prototype in practice:

- JDK - java.lang.Object (through clone() method, classes must implement java.lang.Cloneable interface)
- In combination with the builder pattern - here using just a method like toNewBuilder() for further modifications = deep copy

```

public class Device {
    private String serialNumber;
    private String shortName;
    private final double price;

    public Device(String serialNumber, String shortName, double price) {
        this.serialNumber = serialNumber;
        this.shortName = shortName;
        this.price = price;
    }

    public String getSerialNumber() {
        return serialNumber;
    }

    public String getShortName() {
        return shortName;
    }

    public double getPrice() {
        return price;
    }

    public static class Builder {

        private String serialNumber;
        private String shortName;
        private double price;

        public Builder serialNumber(String serialNumber) {
            this.serialNumber = serialNumber;
            return this;
        }
    }
}

```

```

public Builder shortName(String shortName) {
    this.shortName = shortName;
    return this;
}

public Builder price(double price) {
    this.price = price;
    return this;
}

public Device build() {
    return new Device(serialNumber, shortName, price);
}

}

public Builder toNewBuilder() {
    return new Builder().serialNumber(serialNumber).shortName(shortName).price(price);
}

public static void main(String[] args) {
//when device1 is immutable also due to the final price
    Device device1 = new
Device.Builder().serialNumber("SN123").shortName("router").price(1000d).build();
    System.out.println(device1.price);

    Device device2 = device1.toNewBuilder().price(2500d).build();
    System.out.println(device2.price);
}
}

```

Negatives of creational patterns

- **Abstract factory:** can make code harder to understand.
- **Builder:** shouldn't your class needs to be broken down instead? Дали имаме нужда
- **Dependency injection:** can mask bad design, can introduce temporal coupling - ако инжектираме бийн, който не е инициализиран примерно, ще го разберем късно чак когато пуснем приложението
- **Factory method:** why not composition instead of inheritance?
- **Lazy initialization:** can lead to unpredictable performance
- **Object pool:** leaks, leaks, leaks of resources/memory when too many cache variables
- **Prototype:** really, do your objects need to be so complex?

Singleton: is it a global variable with a fancy name? - например състояние, което може да се сменя от много места от кода заради глобалността, и по този начин е трудно да се контролира

4. 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**

Wrapper Adapter

- Provides a mechanism to ‘adapt’ one incompatible type to another
- Can be used to provide an alternative to multiple inheritance in Java
 - Avoid creating unnecessary objects (An adapter does not need to be created more than once for a given object)

```
public class JuniperRouterAdapter extends CiscoRouter {

    private JuniperRouter juniperRouter;

    public JuniperRouterAdapter(JuniperRouter juniperRouter) {
        this.juniperRouter = juniperRouter;
    }

    @Override
    public void start() {
        juniperRouter.start();
    }

    @Override
    public void stop() {
        juniperRouter.stop();
    }

    public static void main(String[] args) {
        JuniperRouter juniperRouter = new JuniperRouter();
        CiscoRouter ciscoRouter = new JuniperRouterAdapter(juniperRouter);
        ciscoRouter.start();
    }
}
```

Adapter in practice - JDK:

- java.io.InputStreamReader(InputStream)
- java.io.OutputStreamWriter(OutputStream)
- Arrays.asList (although some people think it is a bridge)

Wrapper Bridge

- Allows to decouple abstraction from implementation
- The abstraction and implementation have their hierarchies
- Reduces the number of boilerplate classes that need to be written
- По същество е като на Adapter принципа - едно нещо обвива друго нещо. Даже и кода е почти еднакъв. Разликата е в това какво влагаме

```
public abstract class DeviceController {

    private Device device;

    public DeviceController(Device device) {
        this.device = device;
    }

    public void start() {
        device.start();
    }

    public void stop() {
        device.stop();
    }

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

```

        CiscoRouter device = new CiscoRouter();
        CiscoDeviceController controller = new CiscoDeviceController(device);
        controller.start(); // no need to have CiscoRouterController
    }
}

public class CiscoDeviceController extends DeviceController {
    public CiscoDeviceController(Device device) {
        super(device);
    }
}

public class JuniperDeviceController extends DeviceController {
    public JuniperDeviceController(Device device) {
        super(device);
    }
}

```

Bridge in practice - JDK:

- In AWT the hierarchies of `java.awt.Component` and `java.awt.peer.ComponentPeer`
- `Map.values()`; `Map.keySet()`, `Map.entrySet()` - Map интерфеса не наследява нито `Collection`, нито `Set`

Wrapper Decorator

- An object used to add behavior to another object
- Can be applied in cases when subclassing is not possible or applicable
- Decorator pattern uses typically delegation for the existing operations of the decorated (also called wrapped) object
 - Favor composition over inheritance

```

public abstract class RestartableDevice extends Device {

    private Device device;

    public RestartableDevice(Device device) {
        this.device = device;
    }

    public void restart() {
        device.stop();
        device.start();
    }
}

```

Алтернатива е добавянето на дефолтен метод в интерфейса ако имаме такъв. Т.е. е добре да се помисли дали да се вкарва декоратор или да се използва интерфейс функционалността. Защото ако тези методи за удобство станат много на брой и тежки, тогава не е ок да се ползва интерфейс дефолтни методи.

Добавя нова операция

Decorator in practice - JDK:

- `java.io.BufferedReader/BufferedWriter`
- `Collections.synchronizedCollection`

Wrapper Proxy

- Provides an interface (wrapper) to another object
- Used when the access to the particular object should be controlled
- Can be used to provide additional functionality to an object
- Adapter, Bridge and Decorator are sometimes seen specialization of the Proxy pattern

```
public class CiscoRouterTrackingProxy extends Device {  
    //Добавяме логване  
    private Logger logger = Logger.getLogger(CiscoRouterTrackingProxy.class.getName());  
  
    private CiscoRouter ciscoRouter;  
  
    public CiscoRouterTrackingProxy(CiscoRouter ciscoRouter) {  
        this.ciscoRouter = ciscoRouter;  
    }  
  
    @Override  
    public void start() {  
        logger.info("Starting cisco router ...");  
        ciscoRouter.start();  
    }  
  
    @Override  
    public void stop() {  
        logger.info("Stopping cisco router ...");  
        ciscoRouter.stop();  
    }  
}
```

Променя съществуваща операция (като добавим примерно логване)

Typical use cases for proxy pattern:

- Remote proxy: used to represent an object in a remote system - извикване на метод от един процес/сървър на друг процес/сървър
- Virtual proxy: used to represent a complex or heavy object that cannot be accessed directly
- Protection proxy: used to represent an object that requires access control

Proxy in practice:

- Spring AOP - uses either JDK dynamic proxies (java.lang.reflect.Proxy) or CGLIB to create a proxy for a given target object
- Spring remoting - create proxies for RMI/HTTP/JMS and other invoker classes
- Unit testing frameworks - Mockito mock and spy

Cross-cutting concerns - които не зависят от конкретната реализация/от конкретния код.

Example dynamic proxy:

```
import java.lang.reflect.Proxy;  
  
public class CiscoRouterDynamicProxy {  
    static Device trackingDevice(Device device) {  
        return (Device) Proxy.newProxyInstance(  
            CiscoRouterTrackingProxy.class.getClassLoader(),  
            new Class<?>[]{Device.class},  
            (proxy, method, args) -> {  
                System.out.println("Method: " + method);  
                return method.invoke(device, args);  
            })  
    }  
}
```

```

    );
}

public static void main(String[] args) {
    Device device = new CiscoRouter();
    Device proxy = trackingDevice(device);
    proxy.stop();
    proxy.start();
}
}

```

Facade

- Provides a simpler interface for interacting with a complex system
- Serves as an entrypoint to a particular (sub)system
- Когато искаме да скрием цялата сложност

```

public class DeviceManager {
    private DeviceGroup devices;

    public void initialize() {
        devices = new DeviceGroup();
        // do some complex device initialization ...
        for(Device device : devices.getDevices()) {
            device.start();
        }
    }

    public static void main(String[] args) {
        DeviceManager manager = new DeviceManager();
        manager.initialize();
    }
}

```

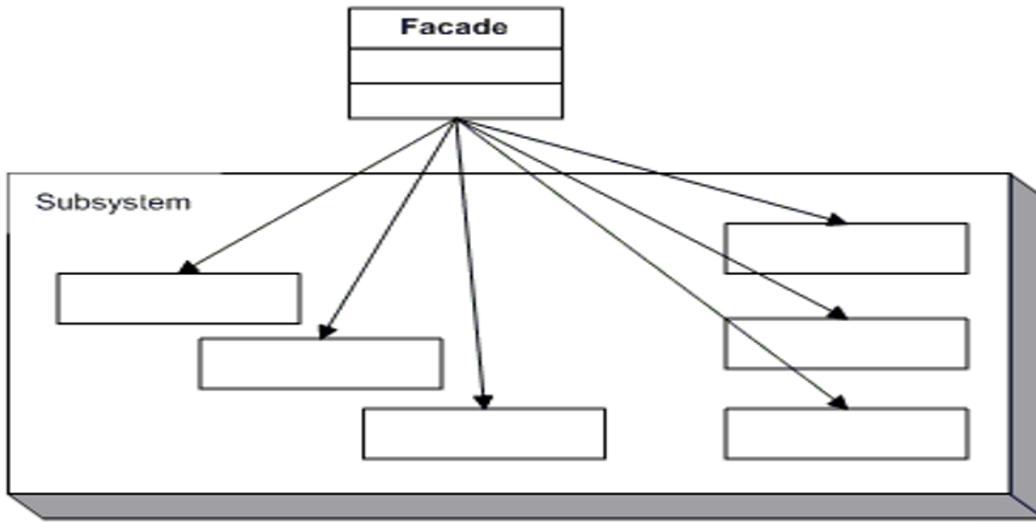
Facade in practice:

- JavaEE - javax.faces.context.FacesContext
- Often applied to hide ugly APIs behind prettier ones - usually just sweeps the problem under the carpet
- Прави се рест апи, което го играе един вид като фасада

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 нова 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");
    }
}
  
```

Front controller

- A common pattern used by web application framework
- Used to handle every request from a client and dispatch accordingly to a proper handler class

```

public abstract class DeviceManager {

    private HashMap<String, Device> devices = new HashMap<String, Device>();

    public abstract Device createDevice(String serialNumber);

    public void addDevice(String serialNumber, Device device) {
        devices.put(serialNumber, device);
    }

}

public class CiscoDeviceManager extends DeviceManager {
    @Override
    public Device createDevice(String serialNumber) {
        Device device = null;
        if(serialNumber.contains("router")) {
            device = new CiscoRouter();
        } else {
            throw new RuntimeException(String.format("Unsupported device: %s", serialNumber));
        }

        return device;
    }
}

public class JuniperDeviceManager extends DeviceManager {

    @Override
    public Device createDevice(String serialNumber) {
        Device device = null;
        if (serialNumber.contains("router")) {
            device = new JuniperRouter();
        } else {
            throw new RuntimeException(String.format("Unsupported device: %s", serialNumber));
        }

        return device;
    }
}

// device controller serves as the front controller
public class DeviceController {
    private HashMap<String, DeviceManager> vendorToDeviceManager = new HashMap<String, DeviceManager>();

    public DeviceController() {
        vendorToDeviceManager.put("cisco", new CiscoDeviceManager());
        vendorToDeviceManager.put("juniper", new JuniperDeviceManager());
    }

    public void invokeOperation(String vendor, String serialNumber, String operation) {
        DeviceManager manager = vendorToDeviceManager.get(vendor); //getting the respective device
manager
        if("create".equals(operation)) {
            manager.createDevice(serialNumber);
        }
    }

    public static void main(String[] args) {
        DeviceController controller = new DeviceController();
        controller.invokeOperation("cisco", "router SN123", "create");
    }
}

```

Front controller in practice:

- Spring - org.springframework.web.servlet.DispatcherServlet
- JavaEE - в JavaEE също има.

Composite interface

- Provides the possibility to compose objects in a **tree structure**
- Treats simple objects and compositions of objects uniformly (равномерно) as trees
- Например директорията също е файл, т.е. малко обръната е логиката

```
public abstract class Device {  
    public abstract void start();  
  
    public abstract void stop();  
  
}  
  
public class DeviceGroup extends Device {  
    private List<Device> devices = new LinkedList<Device>();  
  
    public void addDevice(Device device) {  
        devices.add(device);  
    }  
  
    @Override  
    public void start() {  
        for(Device device : devices) {  
            device.start();  
        }  
    }  
  
    @Override  
    public void stop() {  
        for(Device device : devices) {  
            device.stop();  
        }  
    }  
  
    public List<Device> getDevices() {  
        return devices;  
    }  
}
```

Composite in practice:

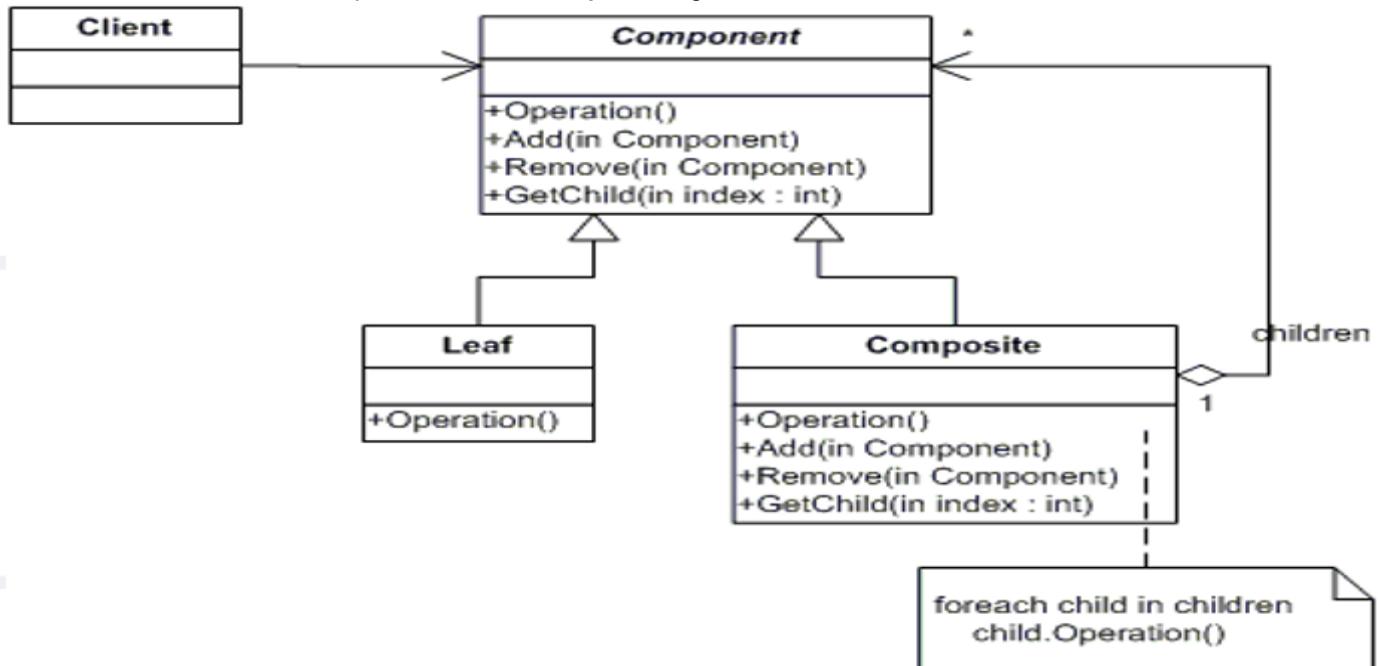
- JDK
 - java.awt.Component
 - java.util.File (indirectly)
- JavaEE
 - javax.faces.component.UIComponent

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));
    }
}

```

Flyweight

- Provides a way to store large objects or large number of objects efficiently - които хабят и много памет
- Avoids creating a large number of objects
- Splits state into:
 - Intrinsic state: heavy, often repeated for each object
 - Extrinsic: unshared, needed for work
 - Да има идентификатор
- Не винаги се използва кеширане, като се каже Flyweight - никога не е едно и също в различни проекти, само идеята е една и съща. Трябва да се види кода и какво точно прави.

```

public class Manufacturer {
    private String name;

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

    public String getName() {
        return name;
    }

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

public class Device {
    private static HashMap<String, Manufacturer> manufacturersCache = new HashMap<>();

```

```

private String serialNumber;

private Manufacturer manufacturer;

public static Device of(String manufacturer, String serialNumber) {
    Device device = new Device();
    device.setSerialNumber(serialNumber);
    Manufacturer manufacturerItem = manufacturersCache.computeIfAbsent(manufacturer,
        (key) -> new Manufacturer(manufacturer));

    device.setManufacturer(manufacturerItem);
    return device;
}

public String getSerialNumber() {
    return serialNumber;
}

public void setSerialNumber(String serialNumber) {
    this.serialNumber = serialNumber;
}

public Manufacturer getManufacturer() {
    return manufacturer;
}

public void setManufacturer(Manufacturer manufacturer) {
    this.manufacturer = manufacturer;
}
}

```

Flyway in practice:

- JDK
 - java.lang.Integer (via valueOf(int)) that caches values in the range of -128 to 127
 - similar behavior for other classes through valueOf() method

Друг пример е електронна таблица или двумерен масив - нито една клетка от таблицата няма нужда да знае на кой индекс по хоризонтала и вертикалата се намира. Самият контекст подава на клетката къде тя се намира. И това е един вид Flyweight.

```

5   public class DeviceGrid {
6
7     Device[][] devices = new Device[][];
8
9
10    void printDevices() {
11        for (int r = 0; r < devices.length; ++r) {
12            for (int c = 0; c < devices[0].length; ++c) {
13                devices[r][c].printDebugInfo(r, c);
14            }
15        }
16    }
17
18 }

```

Marker interface техника

- Provides a mechanism to associate certain metadata with a class
- The metadata is typically used at runtime to determine certain properties of the class

- A runtime annotation can also achieve the same purpose as a marker interface
 - Use marker interfaces to define types

1. Няма методи, а само маркира. Като други класове имплементират маркер Интерфейса

```
public interface RestartableDevice { }
```

2. С Анотации е **новия начин** за реализация - с помоха на Runtime анотации, и чрез reflection гледаме класа анотиран ли е с дадената анотация или не е.

Marker interface in practice:

- JDK
 - java.lang.Cloneable - компроментирано, особености и предупреждения ако се ползва
 - java.io.Serializable - прието е вече, че не е добра практика да се ползва
 - java.rmi.Remote - за извикване на обекти между различни процеси - не се ползва много често
 - java.util.RandomAccess - аз искам тип, който реализира List и RandomAccess интерфейси

5. 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

Chain of responsibility

- Provides the possibility to abstract away command handlers
- Effectively decouples the client from the concrete handler classes
- Typically achieved by creating a sequence of handlers

```
public abstract class DeviceValidator {
    private DeviceValidator next;

    public abstract boolean validate(Device device); //2 implementations

    public DeviceValidator addNext(DeviceValidator validator) {
        next = validator;
        return this;
    }

    public boolean hasNext() {
        return next != null;
    }

    public DeviceValidator getNext() {
        return next;
    }
}

public class PriceValidator extends DeviceValidator {

    @Override
    public boolean validate(Device device) {
```

```

        return device.getPrice() > 0;
    }
}

public class SerialNumberValidator extends DeviceValidator {
    @Override
    public boolean validate(Device device) {
        return device.getSerialNumber().contains("SN");
    }
}

```

Оттук тръгваме с валидацията

```

public class DeviceValidatorChain {

    public boolean validate(DeviceValidator start, Device device) {
        DeviceValidator validator = start;
        boolean valid = true;
        do {
            valid = validator.validate(device);
            validator = validator.getNext();
        } while(valid && validator != null);

        return valid;
    }
}

public class DeviceController {
    public static void main(String[] args) {

        DeviceValidator startValidator = new SerialNumberValidator();
        startValidator.addNext(new PriceValidator());

        DeviceValidatorChain validationChain = new DeviceValidatorChain();
        boolean valid = validationChain.validate(startValidator, new CiscoRouter("SN 123", "router",
1000));
        System.out.println(valid);
    }
}

```

Chain of responsibility in practice:

- JavaEE
 - javax.servlet.Filter (doFilter() methods)

Command pattern

- Provides a mechanism to decouple invoker of a particular operation from the operation itself
- A common interface for command representation is defined by the caller
- Can be implemented as functional interfaces
- Всяко устройство може да си има собствен списък от команди например

```

public abstract class DeviceCommand {
    public abstract void execute(Device device);
}

public class StartCommand extends DeviceCommand {
    @Override

```

```

public void execute(Device device) {
    device.start();
}
}

public class StopCommand extends DeviceCommand {

    @Override
    public void execute(Device device) {
        device.stop();
    }
}

public class DeviceController {

    private HashMap<String, DeviceCommand> commandHandlers = new HashMap<String, DeviceCommand>();

    public void addCommand(String command, DeviceCommand commandHandler) {
        commandHandlers.put(command, commandHandler);
    }

    public void execute(Device device, String operation) {
        DeviceCommand command = commandHandlers.get(operation);
        if(command != null) {
            command.execute(device);
        }
    }

    public static void main(String[] args) {
        DeviceController controller = new DeviceController();
        controller.addCommand("start", new StartCommand());
        controller.addCommand("stop", new StopCommand());

        CiscoRouter router = new CiscoRouter();
        controller.execute(router, "start"); //изпълняваме дадена команда като вземаме от мапа
    }
}

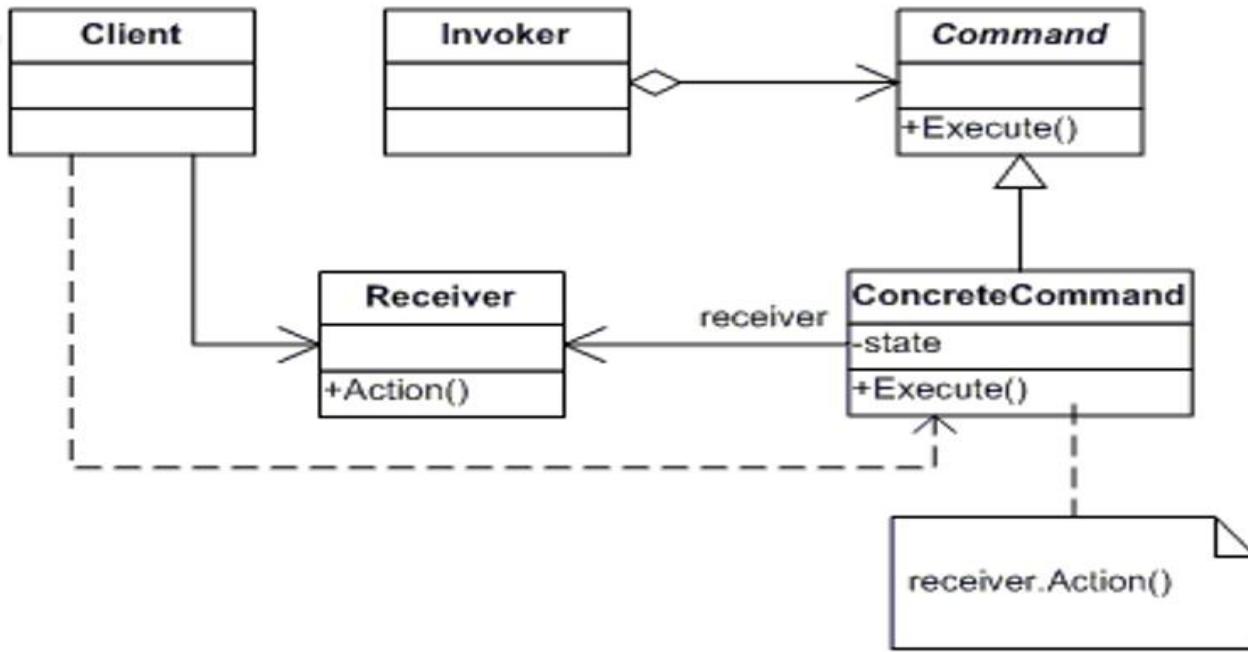
```

Command in practice:

- JDK (Java SE)
 - java.lang.Runnable
 - javax.swing.Action

Command pattern – SoftUni example

- 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());
    }
}

```

Interpreter

- Provides a mechanism to evaluate the grammar of a language
- Each element of the language is “interpreted” by a concrete interpreter class
- The structure of the interpreter classes is organized using the **composite pattern**

```

public class CiscoIOSContext {
    // contains some IOS-related parameters etc.

    private String configurationTarget;

    private String hostname;

    public String getConfigurationTarget() {
        return configurationTarget;
    }

    public void setConfigurationTarget(String configurationTarget) {
        this.configurationTarget = configurationTarget;
    }

    public String getHostname() {
        return hostname;
    }

    public void setHostname(String hostname) {
        this.hostname = hostname;
    }
}

public abstract class CiscoIOSExpression {
    public abstract void execute(CiscoIOSContext context); //3 implementations
}

public class ConfigureCiscoIOSExpression extends CiscoIOSExpression {

    public void execute(CiscoIOSContext context) {
        System.out.println("Configuring...");
        String configurationTarget = context.getConfigurationTarget();
        // execute configure <configurationTarget> ...
    }
}

public class HostnameCiscoIOSExpression extends CiscoIOSExpression {

    public void execute(CiscoIOSContext context) {
        System.out.printf("Hostname: %s\n", hostname);
        String hostname = context.getHostname();
        // execute hostname <hostname> ...
    }
}

```

```

}

public class MultilineCiscoIOSExpression extends CiscoIOSExpression {

    private CiscoIOSExpression[] expressions;

    public MultilineCiscoIOSExpression(CiscoIOSExpression[] expressions) {
        this.expressions = expressions;
    }

    public void execute(CiscoIOSContext context) {
        System.out.printf("Running %d expressions... \n", expressions.length);
        //executes 2 expressions in our case - configure and hostname
        for( CiscoIOSExpression expession : expressions) {
            expression.execute(context);
        }
    }
}

public class CiscoIOSInterpreter {

    public void execute(String script) {

        String[] lines = script.split("\\r?\\n");
        CiscoIOSContext context = new CiscoIOSContext();

        ArrayList<CiscoIOSExpression> expressions = new ArrayList<CiscoIOSExpression>(lines.length);
        for (String line : lines) {
            if (line.startsWith("configure ")) {
                context.setConfigurationTarget(line.replace("configure ", ""));
                expressions.add(new ConfigureCiscoIOSExpression());
            } else if (line.startsWith("hostname ")) {
                context.setHostname(line.replace("hostname ", ""));
                expressions.add(new HostnameCiscoIOSExpression());
            }
        }

        MultilineCiscoIOSExpression multilineExpression = new MultilineCiscoIOSExpression(
            expressions.toArray(new CiscoIOSExpression[0]));
        multilineExpression.execute(context);
    }

    public static void main(String[] args) {

        String script = "configure terminal\n" + "hostname machine.hostname.com";

        CiscoIOSInterpreter interpreter = new CiscoIOSInterpreter();
        interpreter.execute(script);
    }
}

```

Interpreter in practice:

- JDK (Java SE)
 - java.text.Format (DateFormat, MessageFormat, NumberFormat)

Iterator

- Provides a mechanism to access the elements of a composite object sequentially
- Hides specific details on the access mechanism
- Typically used to decouple traversal (обхождане) of collections from the particular collection type

- Итераторът трябва да е отделен клас, който да има референция към колекцията, която ще обхожда. Т.е. итераторът да не променя състоянието на колекцията..!! Под foreach цикъла обхождането стои итератор

За списъци в 99,9% от случаите е добре да използваме готовият `java.util.Iterator<E>` от JavaSE JDK.

```
public abstract class Iterator<T> {

    public abstract boolean hasNext();

    public abstract T next();
}

public class DeviceGroupIterator extends Iterator<Device>{

    private DeviceGroup group;
    private int currentIndex = 0;

    public DeviceGroupIterator(DeviceGroup group) {
        this.group = group;
    }

    @Override
    public boolean hasNext() {
        return currentIndex < group.getDevices().size();
    }

    @Override
    public Device next() {
        return group.getDevices().get(currentIndex++);
    }
}

public class DeviceController {
    public void startCiscoDevices(DeviceGroup deviceGroup) {
        DeviceGroupIterator iterator = new DeviceGroupIterator(deviceGroup);

        while (iterator.hasNext()) {
            iterator.next().start();
        }
    }

    public static void main(String[] args) {
        DeviceGroup group = new DeviceGroup();
        group.addDevice(new CiscoRouter());
        group.addDevice(new JuniperRouter());

        DeviceController controller = new DeviceController();
        controller.startCiscoDevices(group);
    }
}
```

Iterator in practice:

- JDK (Java SE)
 - All implementations of `java.util.Iterator`
 - All implementations of `java.util Enumeration`

Mediator

- Provides a mediator object through which communication between objects happens
- Reduces coupling between objects and simplifies communication

```

public class CiscoDevice {
    private CiscoDeviceManager manager;

    public void start() {}

    public void stop() {}

    public void executeScript(String script) {
        manager.executeScript(script);
    }
}

public class CiscoDeviceManager { //медиатора - всяка комуникация преминава през този мениджър –
    особенно полезен при наличие на обратни callback извиквания на методи

    private CiscoIOSInterpreter interpreter;

    private CiscoDevice ciscoDevice;

    public void setInterpreter(CiscoIOSInterpreter interpreter) {
        this.interpreter = interpreter;
    }

    public void setCiscoDevice(CiscoDevice ciscoDevice) {
        this.ciscoDevice = ciscoDevice;
    }

    public void executeScript(String script) {
        interpreter.execute(script);
    }

    public void startDevice() {
        ciscoDevice.start();
    }

    public void stopDevice() {
        ciscoDevice.start();
    }
}

public class CiscoIOSInterpreter {
    private CiscoDeviceManager manager;

    public void execute(String script) {
        //some code here for executing
    }

    public void startDevice() {
        manager.startDevice();
    }

    public void stopDevice() {
        manager.stopDevice();
    }
}

```

Mediator in practice

- JDK Java SE
 - java.util.concurrent.Executor (the execute() method)

- o `java.util.concurrent.ExecutorService` (the `submit()` method)
- o `javax.swing.ButtonModel`

Memento

- Provides a mechanism to store object's internal state
- In addition to storing, it is also responsible to provide capabilities **for restoring of object's state**

```

public class Device {

    private String serialNumber;
    private String shortName;
    private double price;

    private String configurationScript;

    public Device(String serialNumber, String shortName, double price) {
        this.serialNumber = serialNumber;
        this.shortName = shortName;
        this.price = price;
    }

    public String getSerialNumber() {
        return serialNumber;
    }

    public String getShortName() {
        return shortName;
    }

    public double getPrice() {
        return price;
    }

    public String getConfigurationScript() {
        return configurationScript;
    }

    public void setConfigurationScript(String configurationScript) {
        this.configurationScript = configurationScript;
    }

    public DeviceSnapshot saveConfiguration() {
        DeviceSnapshot snapshot = new DeviceSnapshot();
        snapshot.setConfigurationScript(configurationScript);
        return snapshot;
    }

    public void restoreConfiguration(DeviceSnapshot snapshot) {
        this.configurationScript = snapshot.getConfigurationScript();
    }

    public static void main(String[] args) {
        Device device = new Device("SN 123", "router", 30);

        device.setConfigurationScript("configure terminal\nhostname machine.test.com");
        DeviceSnapshot startSnapshot = device.saveConfiguration();

        device.setConfigurationScript("configure terminal\nhostname another.test.com");
        DeviceSnapshot anotherSnapshot = device.saveConfiguration();

        device.restoreConfiguration(startSnapshot);
        System.out.println(device.getConfigurationScript());
    }
}

```

```

// represents a momento
public class DeviceSnapshot {
    private String configurationScript;

    public String getConfigurationScript() {
        return configurationScript;
    }

    public void setConfigurationScript(String configurationScript) {
        this.configurationScript = configurationScript;
    }
}

```

Memento in practice:

- JDK Java SE - all implementations of java.io.Serializable
- JavaEE - all implementations of javax.faces.component.StateHolder
- Алтернативата на Мементо е да използваме Immutable типове, но по ефективен начин (PersistentDataStructures клас)

Observer

- Provides a mechanism for an object **to notify** a set of dependents (“observers”) **for changes**
- The notifying object is also called a “**source**” and dependents are called “**sinks**”
- Also applied as an architectural concept and provides the building block for distributed event handling systems (such as message brokers)
- Some languages (like C#) provide a built-in support for the observer pattern (Java is not one of them at present)
- Не даваме команда, а назвавме какво се случва
- **onButtonClicked** - думичката on често се използва като част от името
- Publish and Subscribe механизъм е по-сложен от Observer - тъй като при Publish and Subscribe има нанесен и трети обект, който менежира в кой случай какво да прави/къде да препраща. Докато Observer е по-прост - добавяме device switch с add към даден лист, и след това уведомяваме за всеки switch при рестарт примерно.

```

public class Device {

    private List<Device> connectedDevices = new LinkedList<>();

    private String serialNumber;

    private String shortName;

    private double price;

    public Device(String serialNumber, String shortName, double price) {
        this.serialNumber = serialNumber;
        this.shortName = shortName;
        this.price = price;
    }

    public String getSerialNumber() {
        return serialNumber;
    }

    public String getShortName() {
        return shortName;
    }

    public double getPrice() {
        return price;
    }
}

```

```

}

public void addConnectedDevice(Device device) {
    connectedDevices.add(device);
}

public void restartTriggered() {
    // restart current device command...

    for(Device connectedDevice : connectedDevices) {
        connectedDevice.restartTriggered();
    }
}

public static void main(String[] args) {
    Device device = new Device("SN 123", "router", 30);
    Device switch1 = new Device("SN 124", "switch1", 10);
    Device switch2 = new Device("SN 125", "switch2", 10);
    device.addConnectedDevice(switch1);
    device.addConnectedDevice(switch2);

    device.restartTriggered();
}
}

```

Observer in practice:

- JDK Java SE
 - java.util.Observer/java.util.Observable - обикновено искаме някакво по-смислено име и имплементация; **deprecated** - поради лимитации в следене, грешно изпълнение, и т.н. !!!
 - java.util.EventListener
- Spring framework
 - ApplicationContext's event mechanism
- JavaEE
 - servlet listeners

Strategy

- Provides the possibility to vary an algorithm at runtime
- Decouples the client from the concrete algorithm implementation
- A base class provides the abstract method that must be implemented by the concrete implementations provided by the subclasses
- Много често Strategy pattern-а се използва в комбинация с Template method pattern-а

```

public abstract class DeviceValidator {
    public abstract boolean validate(Device device);
}

public class PriceValidator extends DeviceValidator {
    @Override
    public boolean validate(Device device) {
        return device.getPrice() > 0;
    }
}

public class SerialNumberValidator extends DeviceValidator {
    @Override
    public boolean validate(Device device) {
        return device.getSerialNumber().contains("SN");
    }
}

```

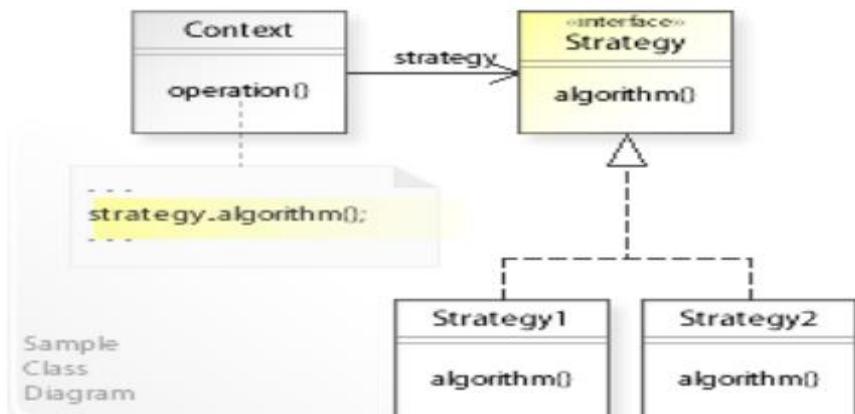
Strategy in practice:

- JDK Java SE
 - java.util.List (the sort() method)
 - java.util.Comparator (the compare() method)

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();
    }
}

```

Template method pattern

- Defines a skeleton method that uses abstract operations to define the behavior of the method
- Can be used in combination with strategy
- A base class provides the template method and subclasses provide implementations
 - Favour the use of standard functional interfaces

```

public abstract class DeviceConfigurationValidator {

    public boolean validate(Device device) {
        boolean result = validateConfigurationSyntax() && validateCommandParameters();
        return result;
    }

    protected abstract boolean validateCommandParameters();

    protected abstract boolean validateConfigurationSyntax();
}

public abstract class CiscoConfigurationValidator extends DeviceConfigurationValidator {

    protected boolean validateConfigurationSyntax() {
        boolean result = true;
        // validate Cisco configuration syntax ...
        return result;
    }

    protected boolean validateCommandParameters() {
        boolean result = true;
        // validate Cisco configuration command parameters ...
        return result;
    }
}

public abstract class JuniperConfigurationValidator extends DeviceConfigurationValidator {

    protected boolean validateConfigurationSyntax() {
        boolean result = true;
        // validate Juniper configuration syntax ...
        return result;
    }

    protected boolean validateCommandParameters() {
        boolean result = true;
        // validate Juniper configuration command parameters ...
        return result;
    }
}

```

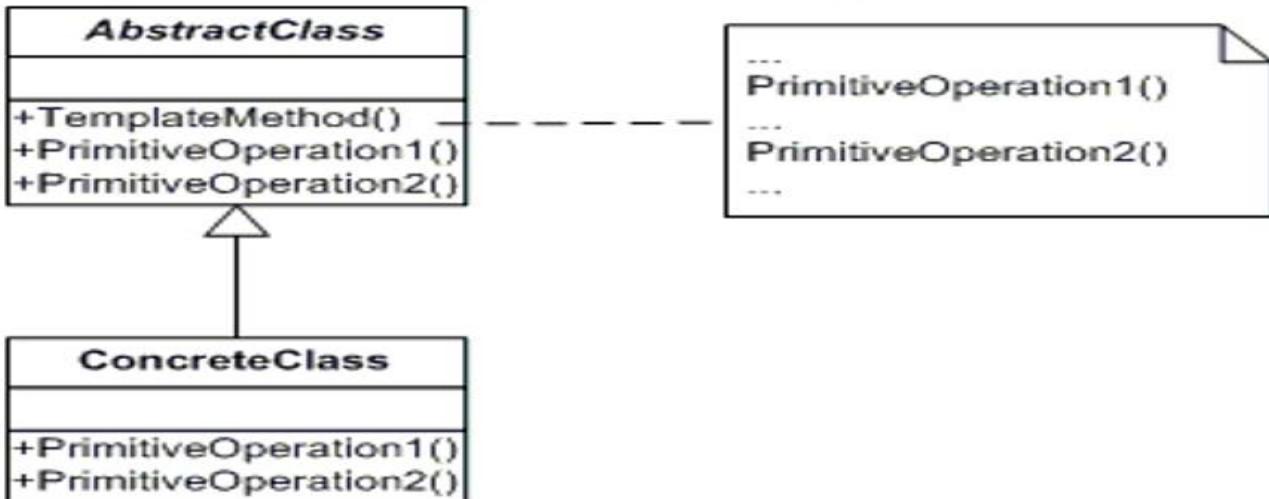
Template method in practice:

- JDK Java SE
 - All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer
 - All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap

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()");
    }
}

```

State

- Provides a mechanism to encapsulate varying behavior for the same object
- Used when an object needs to act differently when its internal state changes

- Can be implemented as a strategy pattern through the state's interface
- Can be implemented also with enum - в Java enum класовете могат да реализират/имплементират интерфейси!!!!

Example with state abstract class

```

public abstract class DeviceState {
    public abstract void restart(); //методът restart() би вършил различни неща в зависимост от
                                    //състоянието
}

public class StartingDeviceState extends DeviceState {
    public void restart() {
        System.out.println("Restarting.....");
        // restart device ...
    }
}

public class StoppedDeviceState extends DeviceState {
    public void restart() {
        System.out.println("Ignoring");
        // ignore ...
    }
}

public class Device {
    private String serialNumber;

    private String shortName;

    private double price;

    private DeviceState state;

    public Device(String serialNumber, String shortName, double price) {
        this.serialNumber = serialNumber;
        this.shortName = shortName;
        this.price = price;
    }

    public void setState(DeviceState state) {
        this.state = state;
    }

    public void restart() {
        state.restart();
    }

    public static void main(String[] args) {
        Device device = new Device("SN1", "device 1", 10.0);
        device.setState(new StartingDeviceState());
        device.restart();

        device.setState(new StoppedDeviceState());
        device.restart();
    }
}

```

Example with enum - недостатък тук е че не може да имаме някакви полета, които да пазят състояние.
Но предимството е че по-видимо, и че са само тези състояния налични.

Example with state enum class

```

public enum EnumDeviceState {
    STARTING{
        @Override
        public void restart() {
            System.out.println("Starting with enum...");
        }
    }
}

```

```

        }
    },
STOPPED{
    @Override
    public void restart() {
        System.out.println("Ignoring with enum");
    }
};

public abstract void restart();
}

public class Device {

    private String serialNumber;
    private String shortName;
    private double price;
    private EnumDeviceState state;

    public Device(String serialNumber, String shortName, double price) {
        this.serialNumber = serialNumber;
        this.shortName = shortName;
        this.price = price;
    }

    public void setState(EnumDeviceState state) {
        this.state = state;
    }

    public void restart() {
        state.restart();
    }

    public static void main(String[] args) {
        Device device = new Device("SN1", "device 1", 10.0);
        device.setState(EnumDeviceState.STARTING);
        device.restart();

        device.setState(EnumDeviceState.STOPPED);
        device.restart();
    }
}
}

```

State in practice:

- JavaEE
 - javax.faces.lifecycle.LifeCycle - the execute() method behaviour is different depending on the current phase of the JSF lifecycle

Visitor

- Provides a mechanism to separate an algorithm from the object structure on which it operates
- Each node in the object structure **can apply an algorithm (visitor)** that is represented by a common interface
- The visitors are typically organized as a **strategy pattern**

```

public class Device {
    private String serialNumber;

    private String shortName;

    private double price;

    public Device(String serialNumber, String shortName, double price) {
        this.serialNumber = serialNumber;
    }
}

```

```

        this.shortName = shortName;
        this.price = price;
    }

    public String getSerialNumber() {
        return serialNumber;
    }

    public String getShortName() {
        return shortName;
    }

    public double getPrice() {
        return price;
    }

    public void validate(DeviceValidator validator) {
        validator.validate(this);
    }

    public static void main(String[] args) {
        Device ciscoRouter = new CiscoRouter("SN 123", "router", 30);
        Device juniperRouter = new JuniperRouter("SN 127", "router", 20);

        DeviceValidator validator = new SerialNumberValidator();
        ciscoRouter.validate(validator);
        juniperRouter.validate(validator);
    }
}

public abstract class DeviceValidator {
    public abstract boolean validate(Device device);
}

public class PriceValidator extends DeviceValidator {
    @Override
    public boolean validate(Device device) {
        return device.getPrice() > 0;
    }
}

public class SerialNumberValidator extends DeviceValidator {
    @Override
    public boolean validate(Device device) {
        return device.getSerialNumber().contains("SN");
    }
}

```

Visitor in practice:

- JDK Java SE
 - javax.lang.model.element.AnnotationValue/AnnotationValueVisitor
 - javax.lang.model.element.Element/ElementVisitor
 - javax.lang.model.TypeMirror/TypeVisitor
 - java.nio.file.FileVisitor/SimpleFileVisitor

38. 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.

39. 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"
}""";
```

40. Object class

In Java, if an object is declared but not initialized, it is null. This is because Java does not automatically assign a default value to objects when they are declared, unlike some other programming languages.

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.

The screenshot shows a Java code editor with the following code:

```

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
    }
}

```

Below the code, a debugger's variable view is shown with the following table:

	Variables
<input type="checkbox"/>	<p>P args = {String[0]@698} []</p> > a = {MySuperObject@700} > b = {MySuperObject@701} f sum = 30 f name = "Pesho" 01 Aa = 1915318863 01 Bb = 1283928880

Handwritten annotations in red:

- Next to the variable list: "референция в паметта"
- Next to the variable list: "hash код"

41. Object cloning

Java SE defines a standard mechanism for cloning of objects

Objects of a class can be cloned if:

- The class implements the **java.lang.Cloneable** marker interface (does not contain any methods)
- Implements properly the `clone()` method defined in the `java.lang.Object` class

Main benefit of object cloning is the possibility to reduce the amount of boilerplate(излишен код) written

Two types of object copies can be created:

- Shallow: only references to the objects are cloned but the underlying data is shared
- Full: the object references and the underlying data is copied by not sharing data between the original and the copy

If the `clone()` method is called on an object that does not implement the `Cloneable` interface a `CloneNotSupportedException` is thrown.

By default the `clone()` method creates a **shallow copy**.

```
@Override  
protected Object clone() throws CloneNotSupportedException {  
    return super.clone();  
}
```

If we want a deep copy, then we can override the method with our own logic like the below:

```
@Override  
public Government clone() {  
    return new Government(primeMinister, ministers);  
}
```

What is the purpose of the `java.lang.Cloneable` interface? - To denote that a class is cloneable

<https://www.digitalocean.com/community/tutorials/java-clone-object-cloning-java>

`CloneNotSupportedException` – if the object's class does not support the `Cloneable` interface. Subclasses that override the `clone` method can also throw this exception to indicate that an instance cannot be cloned.

`throw` - Used to throw an exception for a method, Cannot throw multiple exceptions

`throws` - Used to indicate what exception type may be thrown by a method, Can declare multiple exceptions

Cloning is the process of creating a copy of an Object. Java Object class comes with native `clone()` method that returns the copy of the existing instance. Since Object is the base class in Java, **all objects by default support cloning**.

```
public class Employee implements Cloneable {  
  
    private int id;  
    private String name;  
    private Map<String, String> props;  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getName() {
```

```

        return name;
    }

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

    public Map<String, String> getProps() {
        return props;
    }

    public void setProps(Map<String, String> p) {
        this.props = p;
    }

    // 1. Shallow Cloning
    // The default implementation of Java Object clone() method is using shallow copy.
    // It's using reflection API to create the copy of the instance.
    // The below code snippet showcase the shallow cloning implementation.
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    // 2. Not Shallow Cloning (Deep Cloning)
    @Override
    public Object clone() throws CloneNotSupportedException {

        Employee e = new Employee();
        e.setId(this.id);
        e.setName(this.name);
        e.setProps(this.props);
        return e;
    }

    // 3. Deep Cloning
    // In deep cloning, we have to copy fields one by one. If we have a field with nested
    // objects such as List, Map, etc.
    // then we have to write the code to copy them too one by one. That's why it's called
    // deep cloning or deep copy.
    // We can override the Employee clone method like the following code for deep cloning.
    public Object clone() throws CloneNotSupportedException {

        Object obj = super.clone(); //utilize clone Object method

        Employee emp = (Employee) obj;

        // deep cloning for immutable fields
        emp.setProps(null);
        Map<String, String> hm = new HashMap<>();
        String key;
        Iterator<String> it = this.props.keySet().iterator();
        // Deep Copy of field by field
        while (it.hasNext()) {
            key = it.next();
            hm.put(key, this.props.get(key));
        }
        emp.setProps(hm);

        return emp;
    }
}

```

```
}
```

Пример с изпълнение на clone() метода на подполета на даден клас и с reflection:
условие на задачата – ако полето Object details е от тип, който не се клонира, то да се
хвърли същата грешка каквато ако не можеше да се клонира обект от тип User.

```
public class Main {
    public static void main(String[] args) {
        User user = new User();
        user.setDetails(new Thread());
        user.setDetails(new java.util.LinkedList<Integer>());

        try {
            user.clone();
            System.out.println(true);
        } catch (Exception e) {
            System.out.println(false);
        }
    }
}

public class User implements java.io.Serializable, java.lang.Cloneable {
    private String name;
    private int age;
    private transient Object details;

    public User() {
        this.name = "Svilen";
        this.age = 123;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        User result = (User) super.clone();

        Class<?> aClass = this.details.getClass();
        aClass.getClassLoader();
        java.lang.reflect.Method method;
        try {
            method = aClass.getDeclaredMethod("clone");
        } catch (NoSuchMethodException e) {
            throw new RuntimeException(e);
        }

        try {
            int modifiers = method.getModifiers();
            System.out.println("Modifiers are: " + modifiers);
            method.setAccessible(true);
            //if details not cloneable, then it throws CloneNotSupportedException!!!
            Object resultFromCloningDetails = method.invoke(this.details);
            result.setDetails(resultFromCloningDetails);
        } catch (IllegalAccessException | java.lang.reflect.InvocationTargetException e) {
            throw new RuntimeException(e);
        }

        return result;
    }

    public void setDetails(Object details) {
        this.details = details;
    }
}
```

```
}
```

42. Reduction

Produces single summary by repeatedly applying a combination operation to the elements.

reduce()

Optional<T> **reduce**(BinaryOperator<T> accumulator)

- **Simple, immutable reduction**
- Easy to parallelise

Пример:

```
public class Main {  
    static int x;  
  
    static {  
        x = 10;  
        Stream<Integer> integerStream = // {20, 40, 70, 110}  
            Stream.of(10, 20, 30, 40)  
            .map((y) -> {  
                x += y;  
                return x;  
            });  
  
        System.out.println(x); // 10  
  
        // identity value has not changed. It is still x==10  
        // we sum the numbers  
        int result = integerStream.reduce(x, (a, b) -> {  
            return a + b;  
        });  
  
        // java.util.stream.Stream;  
        // method: T reduce(T identity, BinaryOperator<T> accumulator);  
        // API Note:  
        // Sum, min, max, average, and string concatenation are all special cases of  
        // reduction. Summing a stream of numbers can be expressed as:  
        // Integer sum = integers.reduce(0, (a, b) -> a+b);  
  
        System.out.println(result); // 250  
    }  
  
    public static void main(String[] args) {  
    }  
}
```

collect()

<R, A> R **collect**(Collector<? super T, A, R> collector)

- **Mutable reduction (can keep state)**
- Allows for more sophisticated operations

```
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
import java.time.LocalDate;  
import java.util.List;  
import java.util.Map;
```

```

import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamsDemo {
    record BatteryDay(LocalDate date, String deviceModel, double percentDrained) {
        BatteryDay(String[] fields) {
            this(LocalDate.parse(fields[0]), fields[1], Double.parseDouble(fields[2]));
        }
        BatteryDay(String line) {
            this(line.split(","));
        }
    }

    public static void main(String[] args) {
        try (Stream<String> lines = new BufferedReader(
                new InputStreamReader(
                        StreamsDemo.class.getResourceAsStream("battery.csv")))
                .lines())
        ) {
            // lines.forEach(System.out::println);
            Map<String, List<BatteryDay>> result = lines
                .map(BatteryDay::new)
                .filter(batteryDay -> batteryDay.date.isBefore(LocalDate.now()))
                .collect(Collectors.groupingBy(BatteryDay::deviceModel)); //transform to a
map

            Map<String, Double> result = lines
                .map(BatteryDay::new)
                .filter(batteryDay -> batteryDay.date.isBefore(LocalDate.now()))
                .collect(Collectors.groupingBy(
                    BatteryDay::deviceModel,
                    Collectors.averagingDouble(BatteryDay::percentDrained)
                )); //transform to a map

            System.out.println(result);

            System.out.println(result);
        }
    }
}

```

43. Parsers in Java SE

Parsing libraries for a variety of formats (not only XML and JSON) can be used.

They can be categorized as follows:

- **Low level parsers** where the application needs to convert the parsed format (i.e. XML or JSON) into a proper data structure
- **High level parsers** that map directly the **parsed format into Java objects of a particular class or set of classes**

JAXP - Low level XML parsing

- Low-level parsing capabilities are provided by the **JAXP (Java API for XML Parsing)**
- JAXP provides the following types of XML parsers:
 - o SAX parser: event-driven element-by-element processing – изискват се по-малко памет, тъй като не трябва да се прочете цялото съдържание на XML в паметта
 - o DOM parser: reads the entire model of the parsed XML in memory – прочита цялото съдържание на XML файла в паметта

- StAX parser: event-driven parsing API with simpler model than SAX-based parsers and less memory consumption than DOM-based parsers -

SAX parser

```

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class XMLHandler extends DefaultHandler {

    @Override
    public void startElement(String uri, String localName, String qName, Attributes
attributes) throws SAXException {}

    @Override
    public void endElement(String uri, String localName, String qName) throws
SAXException {}

    @Override
    public void startDocument() throws SAXException {}

    @Override
    public void endDocument() throws SAXException {}
}

```

table.xml

```

<table serialId="123">
    <color>green</color>
    <size>100</size>
    <price>200</price>
</table>

Public static void main(String[] args)
throws ParserConfigurationException, SAXException {

    SAXParserFactory spf = SAXParserFactory.newInstance();
    spf.setNamespaceAware(true);
    SAXParser saxParser = spf.newSAXParser();
    XMLReader xmlReader = saxParser.getXMLReader();
    xmlReader.setContentHandler(new XMLHandler());

    try (FileInputStream fis = new FileInputStream("table.xml")) {
        xmlReader.parse(new InputSource(fis));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

DOM parser

```

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

import javax.xml.parsers.DocumentBuilder;

```

```

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

public void main(String[] args)
throws ParserConfigurationException, IOException, SAXException {

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    DocumentBuilder db = dbf.newDocumentBuilder();
    Document doc = db.parse(new File("table.xml"));
    System.out.println(doc.getFirstChild().getnodeName());

    NodeList children = doc.getFirstChild().getChildNodes();
    for (int i = 0; i < children.getLength(); i++) {
        System.out.println(children.item(i).getNodeName());
        System.out.println(children.item(i).getTextContent());
        System.out.println(children.item(i).getAttributes());
    }
}

```

StAX parser

StAX parsers work in streaming fashion as SAX parsers, but work in **pull manner**(StAX, no handler, we pull elements from the loop one by one)

instead of push manner(SAX) (when events are pushed to a handler)

```

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.XMLEvent;
import java.io.FileNotFoundException;
import java.io.FileReader;

public void main(String[] args) throws FileNotFoundException, XMLStreamException {

    XMLInputFactory factory = XMLInputFactory.newInstance();
    XMLEventReader eventReader = factory.createXMLEventReader(new
FileReader("table.xml"));
    while (eventReader.hasNext()) {
        XMLEvent event = eventReader.nextEvent();
        switch (event.getEventType()) {
            case XMLStreamConstants.START_ELEMENT: {
                System.out.println(event.asStartElement().getName());
            }
        }
    }
}

```

JAXP capabilities

The JAXP API provides a number of additional capabilities such as:

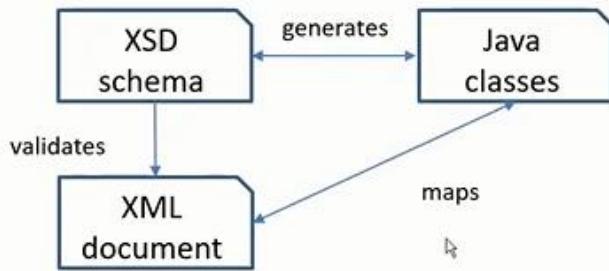
- Validation of the XML documents using **DTDs** (Document Type Definition) schemas and **XSD** (XML Schema Definition) schemas
- Error handling capabilities during XML parsing

- XML traversal capabilities through XPath (which provides path-based domain-specific language for navigation in a XML document)
- XML document transformations using XSLT

JAXB - High level XML parsing

- The JAXB (Java API for XML Binding) API is a higher level API than JAXP and provides the possibility to parse XML documents and map them directly to Java objects
- JAXB classes can be generated from XSD schema
- An XSD schema can be generated from Java classes

Самата XSD схема освен за валидиране на XML документ, то може да се използва и за генериране на Java класове!



За детайлно разписване на примери, виж SpringData записките!

44. Date and time in Java SE(StandardEdition)

- The Java Date and Time API provides classes for working with dates and times in Java
- Before JDK 8, the API was quite limited
- As of JDK 8 a new date and time API based on the **Joda-Time library** is included in the JDK
- Root package of this new Joda-Time library is **java.time**

Има backward compatibility

Pre-JDK 8 API

- Main classes of the Date and Time API (pre-JDK 8) include:

<code>java.util.Date</code>	Represents date and time
<code>java.util.Calendar</code>	Provides arithmetic operation on dates
<code>java.util.GregorianCalendar</code>	Represents Gregorian calendar
<code>java.util.TimeZone</code>	Represents a time zone and provides operations on time zones

- Example (getting the current date)

```
System.currentTimeMillis(); // current date in milliseconds
Date date = new Date(); // current date as a Date instant
```

As of JDK 8 API

- Main classes of the new Date and Time API (as of JDK 8) include:

<code>java.time.Clock</code>	Represents a date in a timezone
<code>java.time.Instant</code>	Represents instantaneous point in time
<code>java.time.Duration</code>	Represents duration
<code>java.time.LocalDate</code>	Represents a date without timezone (ISO-8601)
<code>java.time.LocalTime</code>	Represents a time without timezone (ISO-8601)
<code>java.time.LocalDateTime</code>	Represents a date-time without timezone
<code>java.time.OffsetDateTime</code>	Represents a date time with UTC/Greenwich offset
<code>MonthDay</code>	Month day (ISO-8601)
<code>Year</code>	Year (ISO-8601)
<code>YearMonth</code>	Year-Month (ISO-8601)
<code>ZoneId</code>	A time zone ID such as Europe/Sofia

list: <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>

`java.time.ZonedDateTime` since JDK 1.8

```
Clock clock = Clock.systemUTC();
ZoneId zoneId = ZoneId.systemDefault();
ZoneId customZone = ZoneId.of("Europe/Berlin");
Clock customClock = Clock.system(customZone);
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("MM-DD-YYYY");
LocalDate currentDate = LocalDate.now();
LocalDate localDate = LocalDate.of(2014, Month.JUNE, 10);
localDate = localDate.withYear(2015); //2015-06-10
localDate = localDate.plusMonths(2); //2015-08-10
localDate = localDate.minusDays(1); //2015-08-09
String formattedDate = localDate.format(dateTimeFormatter);

LocalTime localTime = LocalTime.of(20, 30); //hour and minute
localTime = localTime.withSecond(6); //20:30:06
localTime = localTime.plusMinutes(3); //20:33:06

LocalDateTime first = LocalDateTime.of(2014, Month.JUNE, 10, 20, 30);
LocalDateTime second = LocalDateTime.of(localDate, localTime);

//instant is the equivalent of java.util.Date
Instant instant1 = Instant.now();
int monthOfYear = instant1.get(ChronoField.MONTH_OF_YEAR);
Instant addedSixYears = instant1.plus(6, ChronoUnit.YEARS);
Instant instant2 = Instant.now();
Duration duration = Duration.between(instant1, instant2);

ZoneId zone = ZoneId.of("Europe/Paris");
LocalDate date = LocalDate.of(2014, Month.JUNE, 10);
ZonedDateTime zdt1 = date.atStartOfDay(zone);
Instant instant = Instant.now();
ZonedDateTime zdt2 = instant.atZone(zone);
```

- Reasons for the new date and time API in JDK 8:
 - Date is mutable now – може да го променяме, което е добре при многонишково програмиране/няма да дава бъгове
 - Date is not a “date”, but rather an instantaneous point of time – конкретна точка от/във времето

- Dates are constructed with Calendar, but formatters work with Date
- Limited set of operations for working with dates

The new date and time API should be preferred instead of the old one!!!

45. Modules since Java 9 (JDK 9)

Още един начин да групираш нещата, както правиш с пакетите. Но това е ако ти е голяма приложението.

В днешно време пишем малки microservices.

Ако правиш библиотека или framework е по-полезно - помага да зависиш само от това което използваш и прави jar файла по-малък.

46. Locales in Java SE(StandardEdition)

Main info

- Locales are basis for internationalization (i18n) capabilities provided by the Java platform
- Internationalization provides the ability to support multiple languages and regions within an application
- Internationalization extracts text constants from source code so that they can be translated into different languages

To make it internationalized, we need to extract the hardcoded text in a properties file (one per language) and read the text in the corresponding language there.

```
System.out.println(getLocalizedText("BG", "bg"));
```

Messages_<language>_<country>.properties

messages.properties

```
sample_message=Some text in English
```

messages_bg_BG.properties

```
sample_message=Някакъв текст на български
```

```
import java.util.Locale;
import java.util.ResourceBundle;

public static String getLocalizedText(String country, String language) {
    Locale locale = new Locale(language, country);
    ResourceBundle messages = ResourceBundle.getBundle("messages", locale);

    return messages.getString("sample_message");
}
```

- The **java.util.Locale** class is the central unit of internationalization support provided by the Java language
- From the Javadoc: "a Locale object represents a specific geographical, political, or cultural region"
- A **local can be created** in any of the following ways:
 - With the Locale constructors
 - With the Locale.builder class
 - With the Local.forLanguageTag factory method
 - With the constants in the Locale class

```

Locale bgLocale = new Locale.Builder()
    .setLanguage("bg")
    .setRegion("BG")
    .build();

Locale bgLocale = new Locale("bg", "BG");
Locale bgLocale = Locale.forLanguageTag("bg-BG");
Locale deLocale = Locale.GERMAN;

```

- Useful methods of the locale class include:

<code>getAvailableLocales()</code>	Returns an array of available (installed) locales
<code>getISOCountries()</code>	Returns an array of countries as per ISO
<code>getISOLanguages()</code>	Returns an array of languages as per ISO
<code>set()</code> <code>setInt()/setLong() ...</code>	Sets a value for the field on a specified instance. Setter methods are available for fields of primitive type
<code>getCountry()</code> <code>getDisplayCountry()</code>	Returns the country code/localized name
<code>getLanguage()</code> <code>getDisplayLanguage()</code>	Returns the language code/localized name

i18n support

- **Text** is not the only element that may need internationalization support...
- **Numbers and currencies** may be formatted differently according to the country and region
- **Dates and times** can also be formatted differently according to the country and region
- **Custom formats**

Numbers can be formatted according to a locale using the **java.text.NumberFormat** class:

```

int count = 345;
double price = 111.11;
Locale bgLocale = new Locale("bg", "BG");
NumberFormat formatter = NumberFormat.getNumberInstance(bgLocale);
System.out.println(formatter.format(count)); //345
System.out.println(formatter.format(price)); //111,11

```

Formatting currencies – currencies can also be formatted according to a locale using the **java.text.NumberFormat** class:

```

double price = 111.11;
Locale bgLocale = new Locale("bg", "BG");
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance(bgLocale);
System.out.println(currencyFormatter.format(price)); //111,11 лв

```

Formatting dates – dates can be formatted according to a locale using the **java.text.DateFormat** class:

```

Locale bgLocale = new Locale("bg", "BG");
DateFormat formatter = DateFormat.getDateInstance(DateFormat.DEFAULT, bgLocale);
Date today = new Date();
System.out.println(formatter.format(today)); //19.03.2020 г.

```

Formatting times – times can be formatted according to a locale using the **java.text.DateFormat** class:

```

Locale bgLocale = new Locale("bg", "BG");
DateFormat formatter = DateFormat.getTimeInstance(DateFormat.DEFAULT, bgLocale);
Date today = new Date();
System.out.println(formatter.format(today)); //13:55:47 ч.

```

Using custom formats

- Numbers, currencies, dates and times can also be formatted according to a custom pattern and a specified locale.

```
DecimalFormat decimalFormatter = new DecimalFormat("BG###,###.### lv");
System.out.println(decimalFormatter.format(456111.23)); // BG456 111,123 lv
```

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
System.out.println(simpleDateFormat.format(new Date()));
```

DecimalFormat and SimpleDateFormat work also with locales

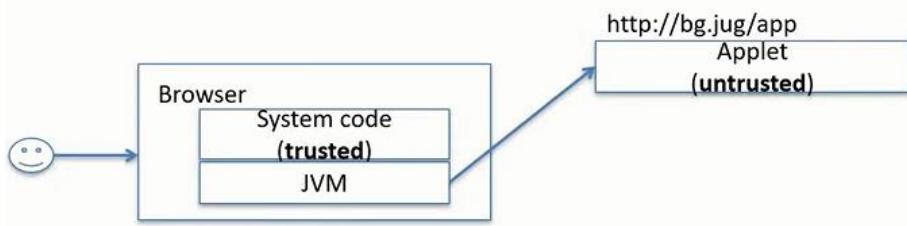
47. Security in Java SE (Standard Edition)

The Java Security model

- Traditionally – companies protect their assets using strict physical and network access policies
- Tools such as anti-virus software, firewalls, IPS/IDS systems facilitate this approach
- With the introduction of various technologies for loading and executing the code on the client machine from the browser (such as Applets – приложения, които могат да се заредят директно в браузъра), a new range of concerns emerge related to client security

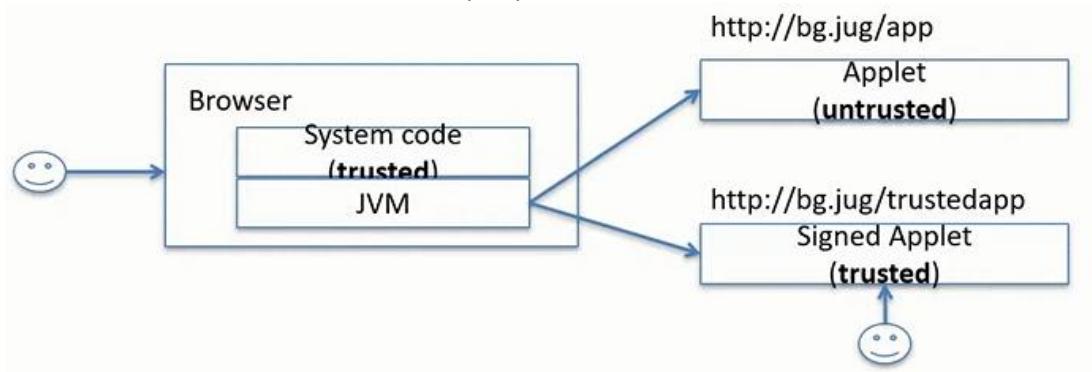
JDK 1.0 (when it all started) – the original security sandbox model was introduced

Всички външни приложения в началото са били untrusted



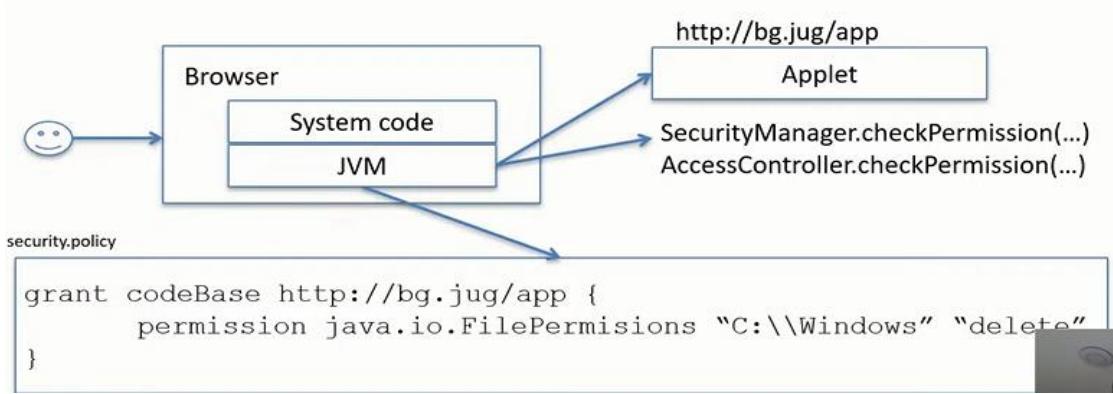
JDK 1.1 (gaining trust) – applet signing introduced

Можем да подпишем с дигитален сертификат



JDK 1.2 (gaining more trust ...) – fine-grained access control

Различни права/роли се добавят. Политиките за сигурност се дефинират от



AccessController предоставя статични методи за проверка на permission-и,
A SecurityManager трябва да го регистрираме при стартиране на JVM

- The notion of **protection domain** introduced – determined by the security policy
- Two types of **protection domains** – **system and application**(приложен)
- The security model becomes code-centric
- No more notion of trusted and untrusted code

JDK 1.3, 1.4 (what about entities running the code ...?) – JAAS

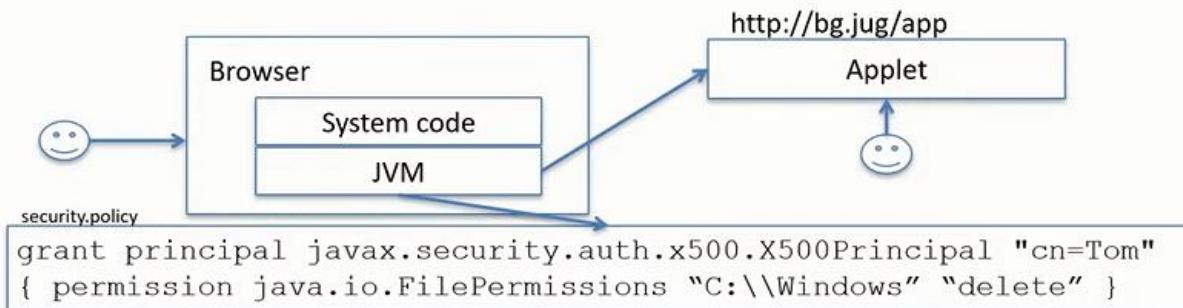
JAAS спецификация – Java Authentication and Authorization Service

Права за:

- това от къде е зареден applet-a,
- но и кой го зарежда.

JAAS разширява security модела в Java като ни позволява permission-те за даден applet да се прилагат и на конкретния потребител.

JAAS API може да аутентицира даден потребител – LDAP сървър, база данни, файлова система, и т.н.



- JAAS extends the security model with **role-based permissions**
- The protection domain of a class now may contain not only the code source and the permissions, but a list of principals
- The authorization component of JAAS extends the syntax of the Java security policy.

Up to JDK 1.4 the following is a typical flow for permission checking:

- 1. Upon system startup a security policy is set and a security manager is installed

`Policy.setPolicy(...)`

`System.setSecurityManager(...)`

- 2. During classloading (e.g. of a remote applet) bytecode verification is done and the protection domain is set for the current classloader (along with the code source, the set of permissions and the set of JAAS principals)
- 3. When **system code** is invoked from the remote code, then the SecurityManager is used to check against intersection of protection domains based on the chain of threads and their callbacks.

```
SocketPermission permission = new
    SocketPermission("jug.bg:8000-9000",
        "connect,accept");
SecurityManager sm = System.getSecurityManager();
if (sm != null) sm.checkPermission(permission);
```

- 4. **Application code** can also do permission checking against remote code using a SecurityManager or an AccessController
- 5. Application code can also do permission checking with all permissions of the calling domain or a particular JAAS subject

```
AccessController.doPrivileged(...)
Subject.doAs(...)
Subject.doAsPrivileged(...)
```

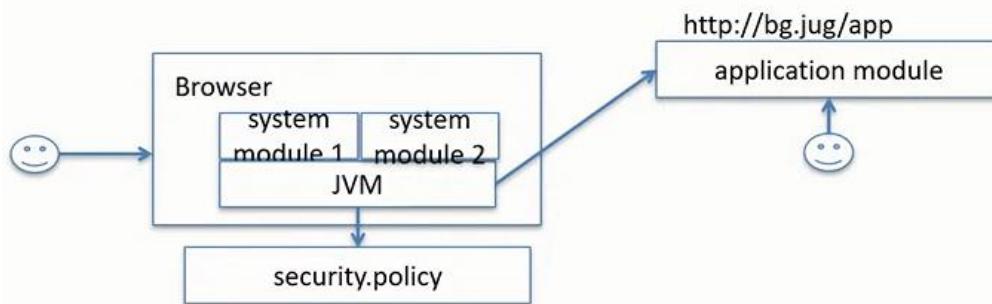
JDK 1.5, 1.6 (enhancing the model ...) – new additions to the security sandbox model (e.g. LDAP for JAAS)

JDK 1.7, 1.8 (further enhancing the model ...)

Enhancements to the sandbox model (e.g. AccessController.doPrivileged() for checking against a subset of permissions)

JDK 1.9 and later (applying the model to modules)

Можем да включваме вече и модули в описанието на правата/permission-ите.



JDK 9, 10 and 11 security enhancements

<https://www.javaadvent.com/2018/12/security-enhancements-in-jdk-9-10-and-11.html>

JDK 12-20 security enhancements

<https://seanjmullan.org/blog/>

JDK 17 – SecurityManager class deprecated for removal

SecurityManager класа е предназначен основно за applet-и.

А самата applet технология е deprecated, дори последните версии на GoogleChrome блокират когато се опитат да заредят JVM.

48. GUI frameworks

The Java platform provides three main technologies for development of **desktop applications**:

- AWT (Abstract Web Toolkit)
- Swing (the successor of AWT)
- **JavaFX**

Of the three, only **JavaFX** is not included as part of the JDK

49. Selenium with Java

General info

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.10.0</version>
</dependency>

import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.WebDriverWait;

@Test
public void shouldContainGoogleResult() throws InterruptedException {
    ChromeDriver chromeDriver = new ChromeDriver();
    WebDriverWait wait = new WebDriverWait(chromeDriver, 10);

    try{
        chromeDriver.get("https://google.com/ncr");

        chromeDriver.executeScript("alert('Hello Selenium')"); //this is JS
        chromeDriver.switchTo().alert().accept();

        chromeDriver.findElement(By.id("L2AGlb")).click();
        chromeDriver.findElement(By.name("q")).sendKeys("jug.bg"); //попълни този текст
        chromeDriver.findElement(By.name("btnK")).click();

        chromeDriver.findElement(By.className("yt-spec-touch-feedback-shape...")).click();

        Thread.sleep(1000); //one way of waiting

        String resultXpath = "//a[@href='https://jug.bg/']";

        //another way of waiting
        WebElement firstResult = wait.until(presenceOfElementLocated(By.xpath(resultXpath)));

        firstResult.click();
    } finally {
        chromeDriver.quit();
    }
}
```

Търсене в конзолата по XPath без да сме инсталирали plugin-a SelectorsHub за XPath:
За Chrome браузър в конзолата на dev-tools:
`$x('//button')` и ни показва колко на брой съвпадения има

```

>>> $x("//div")
=> Array(4) [ div.GoogleActiveViewInnerContainer , div.GoogleActiveViewElement , div.bweb-General-ResponsiveRectangle , div ]
  | 0: <div class="GoogleActiveViewInnerContainer" style="left:0px;top:0px;width:1nts:none;z-index:-9999;">
  | 1: <div class="GoogleActiveViewElement" style="display:inline" data-google-av-cxns="https://pagead2.googlesy...=Cg0A0K7S7BtnglyZAPuEAE" data-google-av-adk="1346364258" data-google-av-metadata="la=0&xi=0& data-google-av-ufs-int
  | google-av-naid="1" data-google-av-slift="" data-google-av-cpnav="" data-google-av-btr="https://securepubads.g.d.m=[UACH]&urlfix=l&adurl=" data-google-av-its="4" data-google-av-flags="["x%2784409ef0f67
  | 2: <div class="bweb-General-ResponsiveRectangle" style="display: block; position_cckground: rgb(0, 0, 0);">
  | 3: <div style="bottom:0;right:0;width:9_!FTkSu0mC'" !important;">
  length: 4
  ><prototype: Array []

```

Page Object model / Page Object pattern

The goal of using page objects is to abstract any page information away from the actual tests. Ideally, you should store all selectors or specific instructions that are unique for a certain page in a page object, so that you still can run your test after you've completely redesigned your page.

Един тестер първо трябва да си опише сценарий по сценарий какво има да тества, и чак тогава да го реализира. Има тулове като Jira, които помагат за това.

```

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

public class YoutubeHomePage {

    private static By LOCATOR_COOKIES_POPUP = By.className("yt-spec-touch-feedback-shape...");
    private static By LOCATOR_ACCEPT_ALL = By.xpath("//span[text()=\"Accept all\"]//...//");
    private static By LOCATOR_SEARCH_BOX = By.xpath("//input[@id=\"search\"]");
    private static By LOCATOR_SEARCH_BUTTON = By.id("search-icon-legacy");
    private WebDriver driver;

    public YoutubeHomePage(WebDriver driver) {
        this.driver = driver;
    }

    public void navigate() {
        driver.get("https://youtube.com/");
    }

    public void acceptCookies() {
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
        wait.until(ExpectedConditions.presenceOfElementLocated(LOCATOR_COOKIES_POPUP));
        driver.findElement(LOCATOR_ACCEPT_ALL).click();
    }

    public void search(String text) throws InterruptedException {
        Thread.sleep(3000);
        driver.findElement(LOCATOR_SEARCH_BOX).sendKeys(text);
        driver.findElement(LOCATOR_SEARCH_BUTTON).click();
    }
}

import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class YoutubeSearchTest {

    @Test
    public void testYoutubeSearch() throws InterruptedException {
        WebDriver driver = new ChromeDriver();
        try {
            YoutubeHomePage youtube = new YoutubeHomePage(driver);
            youtube.navigate();
            youtube.acceptCookies();
            youtube.search("Bulgarian Java User Group");
        } finally {

```

```
        driver.quit();
    }
}
}
```

Cucumber framework and Gherkin

Cucumber ни дава допълнително ниво на абстракция и място за описание при организациите на E2E крайни UI тест функционалност!!!

<https://cucumber.io/>

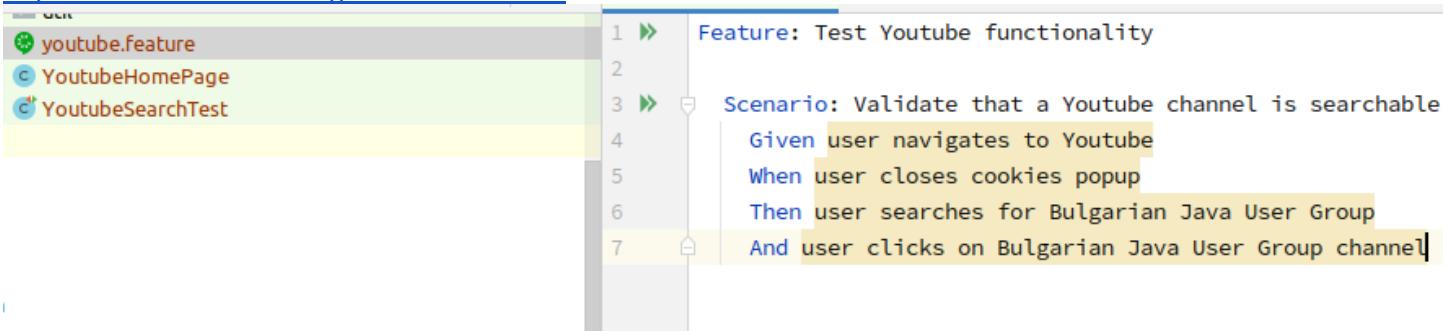
```
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.12.1</version>
</dependency>
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit-platform-engine</artifactId>
    <version>7.12.1</version>
    <scope>test</scope>
</dependency>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${maven-surefire-plugin.version}</version>
        </plugin>
    </plugins>

```

Gherkin syntax

<https://cucumber.io/docs/gherkin/reference/>



Генерираме си и отделен **YoutubeSteps.java** файл като имената на командите от геркин файла съвпадат:

```

public void user_navigates_to_you_tube() {
    // Write code here that turns the phrase above into concrete actions
}
@When("user closes cookies popup")
public void user_closes_cookies_popup() {
    // Write code here that turns the phrase above into concrete actions
}
@Then("user searches for Bulgarian Java User Group")
public void user_searches_for_bulgarian_java_user_group() {
    // Write code here that turns the phrase above into concrete actions
}
@Then("user clicks on Bulgarian Java User Group channel")
public void user_clicks_on_bulgarian_java_user_group_channel() {
    // Write code here that turns the phrase above into concrete actions
}
private YoutubeSteps youtubePage,
publicYoutubeSteps() {
    driver = new ChromeDriver();
    youtubePage = new YoutubeHomePage(driver);
}

@Given("user navigates to YouTube")
public void user_navigates_to_you_tube() {
    youtubePage.Navigate();
}

@When("user closes cookies popup")
public void user_closes_cookies_popup() {
    youtubePage.acceptCookies();
}

```

```

import org.junit.platform.suite.api.Suite;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.IncludeEngines;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
public class CucumberTest {
}

```

И накрая пускаме тази конзолна команда:
mvn clean test

SikuliX

SikuliX - специална библиотека, която на базата на скрийншот търси по дом дървото даден елемент и го клика/попълва и т.н.

50. Многонишково програмиране (Multithreading) в Java(SE)

Threads

- Threads are also referred to as **lightweight processes**
- Threads are separate execution flows within a single process
- They share the resources of the process including open files and memory

- Java threads are typically bound to native OS threads that execute them on different OS processors
- Some JVM implementations of threads also provide so called **green threads** that are scheduled for execution by the JVM itself rather than the OS - от кого се изпълняват
- Threads in the JVM might be
 - user потребителски
 - or daemon threads - отделни независими нишки необвързани с друга нишка от JVM, примерно за scheduling

Which of the following **is not valid** about Java threads?

- a.every time a thread is created a new stack is allocated
- b.threads have different states
- c.threads can have a priority
- d. Java threads are not bound to OS threads

What are **lightweight threads** in Java?

- a. **scheduled only by the JVM and use less resources**
- b. easier to schedule native threads
- c. easier to create than standard Java threads
- d. standard Java threads with limited capabilities

Когато пуснем едно стандартно Java приложение:

- A typical Java application contains multiple threads of execution
- The JVM itself once started starts different threads - например нишки за Garbage collector-а, които нишки са native (не минава през предоставените от самата JVM средства за работа с нишки)
- The Java application may also start a number of threads during its execution
- Има готови пакети/класове/библиотеки (като **java.util.concurrent**) - които позволяват по лесен и удобен начин да работим с нишки

Green threads, normal threads, virtual threads

Green Threads(old) had an N:1 mapping with OS Threads. All the Green Threads ran on a single OS Thread.
With Virtual Threads, multiple virtual threads can run on multiple native carrier OS threads (n:m mapping)
The old Java green threads could only use a single core, the new Java virtual threads can use multiple cores.

Like with green threads, virtual threads are managed by the JVM!

The most lightweight version of threads are **fibers(virtual threads)** that are parallel execution flows within a single thread (като тези виртуални нишки са в рамките само на JVM, без да се взаимодейства с RAM паметта на операционната система (но реално carrier-а е нишката на ОС). По-лесни са за менъджиране, но нямат пълният набор от опции като стандартната нишка). Currently native support for fibers(virtual threads) in the JVM is being developed under the **project Loom**. От версия **JDK 20/21** може да се използват вече.

In computer programming, a **green thread** (virtual thread) is a thread that is scheduled by a runtime library or virtual machine (VM) instead of natively by the underlying operating system (OS). Green threads emulate multithreaded environments without relying on any native OS abilities, and they are managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support.

Единствената разлика между VirtualThreads и GreenThreads е в това, че VirtualThreads използват **Thread carriers**, а другите GreenThreads не.

Virtual threads are a lightweight implementation of threads that is provided by the JDK rather than the OS. They are a form of user-mode threads, which have been successful in other multithreaded languages (e.g., goroutines

in Go and processes in Erlang). User-mode threads even featured as so-called "green threads" in early versions of Java, when OS threads were not yet mature and widespread. However, Java's green threads all shared one OS thread (M:1 scheduling) and were eventually outperformed by platform threads, implemented as wrappers for OS threads (1:1 scheduling). Virtual threads employ M:N scheduling, where a large number (M) of virtual threads is scheduled to run on a smaller number (N) of OS threads.

When Quarkus meets Virtual Threads

<https://quarkus.io/blog/virtual-thread-1/>

At the beginning of the Java time, Java had *green threads*. Green threads were user-level threads scheduled by the Java virtual machine (JVM) instead of natively by the underlying operating system (OS). They emulated multithreaded environments without relying on native OS abilities. They were managed in user space instead of kernel space, enabling them to work in environments that do not have native thread support. Green threads were briefly available in Java between 1997 and 2000. I used green threads; they did not leave me with a fantastic memory.

In Java 1.3, released in 2000, Java made a big step forward and started integrating **OS threads**. So, the threads are managed by the operating system. It is still the model we are using today. Each time a Java application creates a thread, a platform thread is created, which wraps an OS thread. So, creating a platform thread creates an OS thread, and **blocking a platform thread blocks an OS thread**.

Java 19 introduced a new type of thread: virtual threads. In Java 21, this API became generally available.

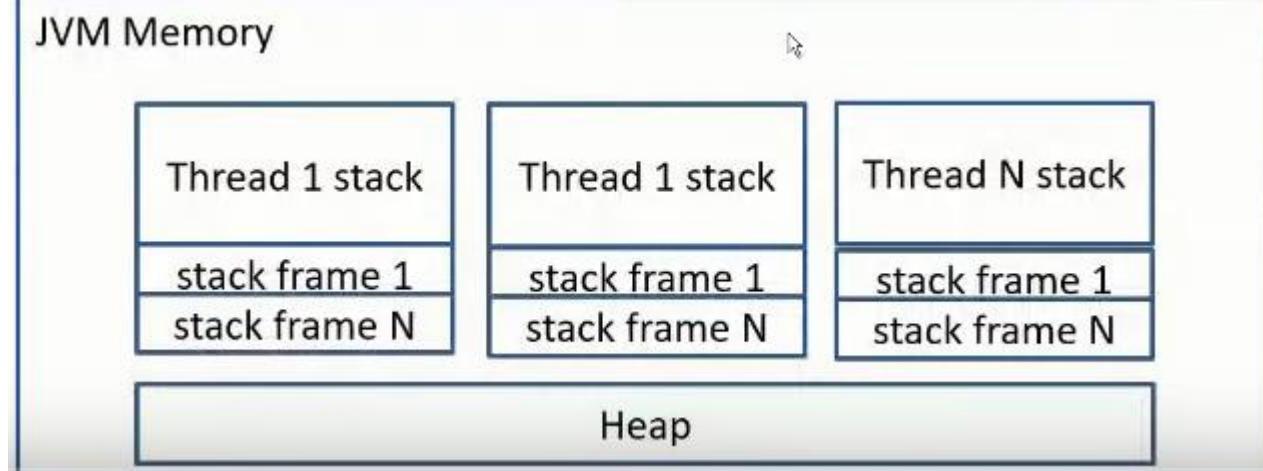
The Java Memory Model (JMM)

- The Java Memory Model describes how threads interact with the JVM memory
- Different processors typically provide **local caches** that are synchronized with **main memory**
- Since Java threads may run on different processors they may see a different view of the same shared memory due to the caches

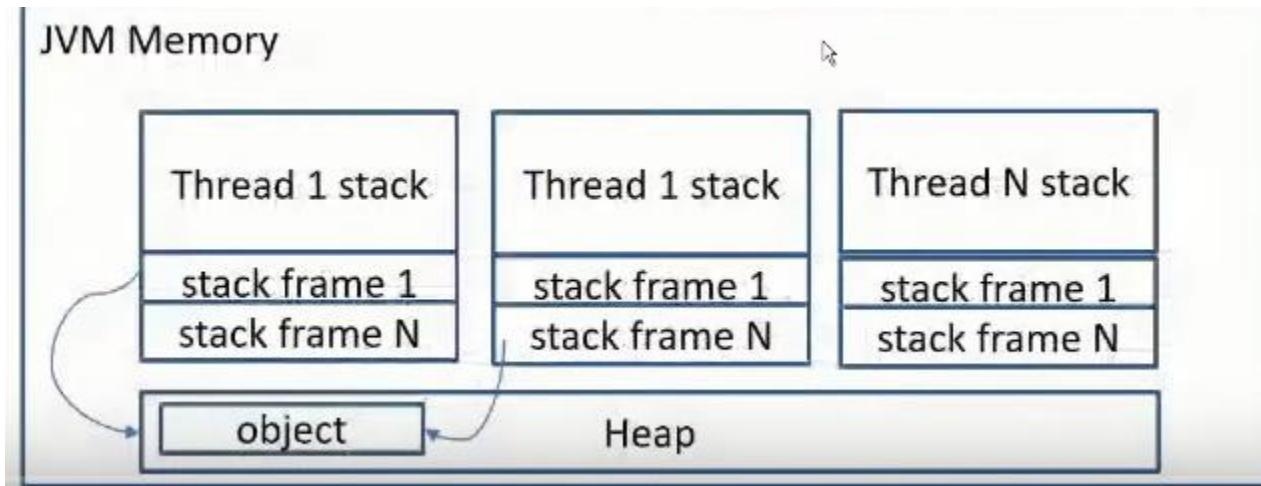
Keeping processors caches at all time in sync with main memory is costly and not always feasible (и не винаги възможно) due to performance penalties.

Или да виждат само локалния кеш или всичко да е синхронизирано с основната памет.

- Each thread has its own **thread stack** allocated - стак на изпълнение, който се пълни със стак фреймове
- The thread stack contains local variables that are not visible to other threads



- Since objects are stored on the heap they might be accessed by multiple threads
- Additional mechanisms should be used to ensure proper memory visibility and synchronization among threads - с други думи ако правим някаква промяна на стойност на поле на обект от heap-а от една нишка, то тази промяна е видима за останалите нишки!



The **JVM memory (stack and heap)** is mapped to the RAM on the target machine.

- Synchronization with the main memory can be achieved as follows:
 - Marking a variable as **volatile** makes reading from and writing to the variable directly from main memory - може да сложим тази запазена дума на поле на клас и тогава сме синхронизирани с основната РАМ памет
 - Creating **synchronized block** with the **synchronized** keyword may also perform a cache flush (in most cases partial using special instruction like memory barriers) once a thread exits the block - т.е. ако дадена нишка е влязла в блока да изпълнява кода, то други нишки опитващи се да достъпят същия блок (public void ...) изчакват.

Synchronized blocks in that regard serve not only as a mechanism to provide order of execution, but also memory visibility.

Всеки обект в JVM има поле **lock**, по което поле дадена нишка може да го достъпи и да го заключи.

volatile - това че сме синхронизирани с основната РАМ памет ни предпазва да не върнем друга стойност на дадената променлива.

Threads and thread pools

Java threads

There are two main ways to create a thread:

- By implementing the **java.lang.Runnable** interface (preferred way)

```
new Thread(new Runnable() {
    @Override
    public void run() {
        //thread logic here
    }
}).start();
```

```
new Thread(() -> {
    //thread logic here
}).start();
```

- By extending the **java.lang.Thread** class - изменя поведението на самия Thread клас, което не е добре

```
public class CustomThread extends Thread {
    @Override
    public void run() {
        //thread logic here
    }
}
```

```
}
```

```
new CustomThread().start();
```

When is a thread scheduled for execution by the JVM?

- a.when calling the run() method
- b.when calling the start() method**
- c.when the thread is initialized
- d.it is not bound to a method invocation

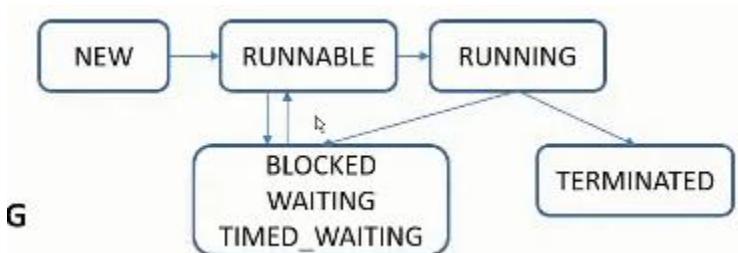
When we call **start()** with an object of **Thread** class that thread goes to the **Runnable** state. So all the threads go to **Runnable** state after calling **start()** by the object of those threads. It is **JVM thread scheduler**, who picks thread randomly from **Runnable** state to give it in **Running** state. After going to **Running** state, the determined call stack for that specific thread becomes executed.

Again, JVM thread scheduler can stop the execution of a thread by picking that thread from Running state to Runnable state. This time, code execution is paused in the call stack of that thread.

Thread states

A non-running thread might have any of the states defined in the Thread#State enum

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIME_WAITING
- TERMINATED



Thread interruption

- A thread can be interrupted by calling the **Thread.interrupt** method on the thread object
- The **Thread.interrupt** method sets a special flag that indicates the thread is interrupted and that can be checked with the **Thread.isInterrupted** method on the thread object.
- Some methods such as a Thread.sleep throw a **java.lang.InterruptedIOException** if the calling thread is interrupted.

Thread pools

- Threads might be **reused** since thread creation is an expensive process
- In the JDK different thread pools are provided as an implementation of the **java.util.concurrent.Executor** interface
- The **java.util.concurrent.Executors** class provides static methods for the creation of **different types of thread pools** - Например при процесор с 16 ядра, то се създават 16 нишки за примерно 1500

различни задачи. Няма смисъл да се правят повече нишки, защото тогава ще има много **context switching**, който е много скъп процес!

- The **java.util.concurrent.Executor** interface is extended by an **ExecutorService** interface that provides more operations for thread pools
- The **ExecutorService** interface is further extended by the **ScheduledExecutorService** interface that provides the possibility to schedule threads for execution

Съществуват различни варианти за работа с pools:

- The **Executors** class and **Executor** interface
- The **ExecutorService**
- **ScheduledExecutorService** – extends **ExecutorService**
- The **ThreadPoolExecutor**
- The **ForkJoinPool**

<https://stackify.com/java-thread-pools/>

```
// final int numThreads = 10;
int numThreads = Runtime.getRuntime().availableProcessors(); //8
ExecutorService threadPool = Executors.newFixedThreadPool(numThreads);

for (int i = 0; i < 100; i++) {
    Runnable task = new Task(i); //Task класа е имплементация на java.lang.Runnable
    threadPool.execute(task);
}

threadPool.shutdown();

// waits for all threads to finish
threadPool.awaitTermination(2, TimeUnit.SECONDS);

System.out.println("Finished all threads");
```

Which of the following **is a not** a thread pool characteristic ?

- a.Optimizes the creation and scheduling of threads
- b.Provide a mechanism to execute a task in a distributed manner
- c.Improves resource utilization in the application
- d. **Eliminates the presence of deadlocks**

Which of the following **is not a valid type of thread pool** as defined by the **java.util.concurrent.Executors** class?

- a. cached thread pool - `java.util.concurrent.Executors.newCachedThreadPool();`
- b. single thread pool - `java.util.concurrent.Executors.newSingleThreadExecutor();`
- c. scheduled thread pool - `java.util.concurrent.Executors.newScheduledThreadPool();`
- d. native thread pool

Example with ExecutorService

```
ExecutorService service = Executors.newCachedThreadPool();
```

```
Runnable r1 = () -> transferAmount(new PaymentIqTransferDTO(USER_ID, "100.00", EUR,
TX_ONE_ID, "001", "Deposit", "Bambora",
"101", "EUR", "1.00", "EUR", "11111A1", "A", null));
Runnable r2 = () -> transferAmount(new PaymentIqTransferDTO(USER_ID, "200.00", EUR,
TX_TWO_ID, "001", "Deposit", "Bambora",
"201", "EUR", "1.00", "EUR", "11111A1", "A", null));
```

```

// Submits a Runnable task for execution and returns a Future representing that task. The Future's get method will return null upon successful completion.
Future<?> future = service.submit(r1);
service.submit(r2);

// Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.
service.awaitTermination(1000, TimeUnit.MILLISECONDS);

```

Thread locals

- Thread locals are special types of variables that can only be written and read from a single thread - т.е. паметта на такава нишка не се споделя с други нишки

```

private ThreadLocal threadLocal = new ThreadLocal<>();
threadLocal.set("some value");
String value = (String) threadLocal.get();

```

Futures

- Futures are used to represent the result of a future execution
- In Java futures are represented by the **java.util.concurrent.Future** interface
- Futures provide the possibility to **cancel** a task, **check** if the task is **done** or **get the result** (blocks until computation is done) - позволява ни да върнем и резултат от задачата

```

final int numThreads = 10;
ExecutorService executor = Executors.newFixedThreadPool(numThreads);

Runnable task = new Task();
Future<String> future = executor.submit(task);

// blocks until task is completed.
//In the case of using Runnable, it executes the overridden run() method.
//In the case of using Callable, it executes the overridden call() method.
String result = future.get();

```

Callable vs Runnable

In a **java.util.concurrent.Callable** interface that basically **throws a checked exception and returns some results**.

This is one of the major differences between the upcoming **java.lang.Runnable** interface **where no value is being returned**. In this **Runnable** interface, it simply computes a result else throws an exception if unable to do so.

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

What is a difference between a Callable and a Runnable?

- Callable can throw a checked exception while a Runnable cannot
- Callable cannot be scheduled by a thread pool while a Runnable can
- Runnable cannot be scheduled by a thread pool while a Callable can
- Callable can be canceled while Runnable cannot

Example Fibonacci with Futures:

```
public class Main {
    public static void main(String[] args) {
        java.util.concurrent.ExecutorService pool =
            java.util.concurrent.Executors.newSingleThreadExecutor();

        try {
            java.util.concurrent.Future<java.math.BigDecimal> result = fact(1, pool);
            java.util.concurrent.Future<java.math.BigDecimal> result = fact(5, pool);

            System.out.println(result.get()); //the get() methods executes the call() method
which go into recursion and returns the result
        } catch (Exception ex) {
        } finally {
            try {
                pool.awaitTermination(1, java.util.concurrent.TimeUnit.SECONDS);
            } catch (InterruptedException e) {
            }
        }

        pool.shutdown();
    }
}

public static java.util.concurrent.Future<java.math.BigDecimal> fact(int n,
java.util.concurrent.ExecutorService pool) {

    Multiply previousMultiply = new Multiply(null, java.math.BigDecimal.ONE);
    java.util.concurrent.Future<java.math.BigDecimal> lastTaskResult =
pool.submit(previousMultiply);

    for (int start = 2; start <= n; start++) {
        Multiply current = new Multiply(previousMultiply,
java.math.BigDecimal(start));
        lastTaskResult = pool.submit(current);
        previousMultiply = current;
    }

    return lastTaskResult;
}
}

public class Multiply implements java.util.concurrent.Callable<java.math.BigDecimal> {
    private Multiply start;
    private java.math.BigDecimal end;

    public Multiply(Multiply start, java.math.BigDecimal end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public java.math.BigDecimal call() {
        if (this.start == null) {
            return java.math.BigDecimal.ONE;
        }

        java.math.BigDecimal startResult = this.start.call();
        return startResult.multiply(this.end);
    }
}
```

```
}
```

Thread synchronization

Info

- Since different threads can access shared data this may result in data consistency issues also known as **race condition**
- To solve issues related to race conditions a mechanism called thread synchronization can be used
- Java (JVM) uses object monitors to perform thread synchronization
- A monitor is associated with an object and conducts a region that can be accessed by one thread at a time - благодарение на lock се задава регион
- A monitor is acquired by a thread using a **lock**
- A lock can be either **intrinsic** or **extrinsic**

Intrinsic locks

An **intrinsic** block is created using the **synchronized** keyword that:

- can be applied as a method attribute that makes the method block a synchronized region using the method's object as a monitor

```
public synchronized void someMethod() {}
```

- can be specified as a separate block within a method using a target object as a monitor

```
synchronized (object) {  
    //block of code  
}
```

What is 'synchronized' keyword used for ?

a. To define an implicit lock on a **object**

b. To define an implicit lock on a thread

c. To define an implicit lock on a method

d. To define an implicit lock on a block

Extrinsic locks

- Extrinsic locks are provided as different implementations of the **java.util.concurrent.locks.Lock** interface:
 - **ReentrantLock** - това е по подразбиране за **intrinsic locks** - ако дадена нишка държи lock към някакъв обект, и влеземе например в друг блок/метод, който държи същия lock, тогава нишката успява да влезе в него
 - **ReadWriteLock**
 - **StampedLock**
- Extrinsic locks provide more flexibility than intrinsic locks such as:
 - The possibility to create the synchronized block across multiple methods
 - The possibility to check if a lock is already acquired with the **tryLock** method

Same as having **synchronized** on the **this** object

```
public void someMethod() {  
    reentrantLock.lock();  
    try {  
        // block of code  
    } finally {  
        reentrantLock.unlock();  
    }  
}
```

```
}
```

Joining threads – making order

- The **Thread.join** method provides a mechanism whereby one thread can wait for the completion of another thread - т.е. ни дава възможност да chain-ваме в ПОРЕДНОСТ изпълнението на дадени НИШКИ

```
// called by thread A
public void someMethod() {
    // some logic ...

    // wait for thread B to complete
    threadB.join();
}
```

```
thread1.start();
thread2.start();
thread1.join();
thread2.join();
```

Synchronization issues

- Problems may occur when synchronization is not applied properly such as:
 - deadlock:** two or more threads are locked indefinitely waiting for each other

//нишка 1 локва обект o1, и след като мине 1 секунда, тръгва да вика обект o2. Но обект o2 в това време е локнат от нишка 2!!!

```
public class DeadlockExample {
    public static void main(String[] args) {
        Object o1 = new Object();
        Object o2 = new Object();

        Thread thread1 = new Thread(() -> {
            synchronized (o1) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                synchronized (o2) {
                    System.out.println("Hello from Thread 1");
                }
            }
        });
    }

    Thread thread2 = new Thread(() -> {
        synchronized (o2) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            synchronized (o1) {
                System.out.println("Hello from Thread 2");
            }
        }
    });
}
```

```

        }
    });

    thread1.start();
    thread2.start();
}
}

```

- **starvation:** a thread is not able to gain access to shared resource thus not being able to make progress
- **livelock:** similar to deadlock but occurs when two or more threads act on each other as a response and are not blocked but continue indefinitely without being able to make progress

Примери

```
System.out.println(thread1.getState());
```

```

thread1.setPriority();
thread1.isDaemon();
thread1.isInterrupted();
thread1.getThreadGroup(); - ако искаме да копираме дадени нишки и да можем да ги
менъжираме след това
thread1.setName();
thread1.sleep(4000);

```

```
Thread.currentThread().sleep(4000)
```

Thread safety

Thread safety

- Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly
- Immutable objects are thread safe
- To make a class immutable the following can be done:
 - mark all class fields final or declare the class as final
 - ensure **this** reference is not allowed to escape during construction
 - make any fields which refer to mutable data objects private
 - don't provide setter method

Java monitor pattern

Само една нишка може да достъпи дадения метод в даден момент. Това може да е проблем ако заключваме целия метод. Вместо това можем да синхронизираме дадена част от въпросния метод където има критични секции – т.н. compound actions – някакъв брояч например за който само е задължително **synchronized**.

Или пък да използваме **AtomicInteger** вместо това – дава синхронизация дефакто.

Критични секции в многонишковото програмиране са всичко, което е **mutable!** В света на Java – слагаме **final** думичката правейки го **immutable**, което ни гарантира initialization safety в многонишкова среда! Важна част в цялото нещо заема и encapsulation-a!

```
public class ValidNamesService {

    private final Object lock = new Object();

    private final Set<String> validNames;

    public ValidNamesService() { this.validNames = new HashSet<>(); }

    public void addName(final String name) {
        synchronized (lock) {
            this.validNames.add(name);
        }
    }

    public boolean hasName(final String name) {
        synchronized (lock) {
            return this.validNames.contains(name);
        }
    }
}
```

Reordering

При работа с нишки, понякога JVM прави автоматичен re-ordering и може да се получи разместване кое след кое се изпълнява. Тогава се налага да използваме locks!

```
private static void testMethod() {
    number = 0;
    isFinished = false;
    final Thread firstThread = new Thread(() -> {
        while(!isFinished) {
        }
        System.out.println(number);
    });
    firstThread.start();
    number = 42;
    isFinished = true;
}
```

```
0
42
42
0
0
42
42
42
42
....
```

Publishing and escaping

Escaping става на въпрос когато работим в стека на дадена нишка, то да не се обръщаме/да публикуваме/ към методи извън стека на нишката. Ако се обръщаме, то ни трябва експлицитно да зададем lock! Иначе се получава **escaping**.

```
private static List<List<Integer>> cache = Collections.synchronizedList(new ArrayList<>());  
  
public static void main(final String... args) {  
    // Numbers are confined to stack of the thread  
    final Thread thread = new Thread(() -> {  
        final int number = 1;  
        final List<Integer> numbers = new ArrayList<>();  
        numbers.add(number);  
        System.out.println(numbers);  
        // Numbers are now escaping  
        // Avoid doing this  
        cache.add(numbers);  
    });  
    thread.start();  
}
```

Confinements

Че използваме методи предназначени само за стека на дадената нишка

Thread confinement

Един thread се занимава само с numbers в случая

```
public class ThreadConfinement {  
    private static final List<Integer> numbers = new ArrayList<>();  
  
    public static void main(final String... args) {  
        // Numbers are confined to thread  
        final Thread thread = new Thread(() -> {  
            numbers.add(1);  
            System.out.println(numbers);  
        });  
        final Thread secondThread = new Thread(() -> {  
            System.out.println("Second thread");  
        });  
        thread.start();  
    }  
}
```

AdHoc confinement

.....

Stack confinement

Да не го публикуваме в cache един вид, защото е извън scope на стека на нишката.

```
private static List<List<Integer>> cache = Collections.synchronizedList(new ArrayList<>());  
  
public static void main(final String... args) {  
    // Numbers are confined to stack of the thread  
    final Thread thread = new Thread(() -> {  
        final int number = 1;  
        final List<Integer> numbers = new ArrayList<>();  
        numbers.add(number);  
        System.out.println(numbers);  
        // Numbers are now escaping  
        // Avoid doing this  
        cache.add(numbers);  
    });  
    thread.start();  
}
```

Synchronized collections

Всеки един метод на тези колекции е синхронизиран също.

- **StringBuffer** is a thread-safe alternative of **StringBuilder**
- The **java.util.Collections** class provides methods to retrieve thread-safe collections
 - **Collections.synchronizedSet()**
 - Synchronized list - **Collections.synchronizedList()**
 - Synchronized map - - **Collections.synchronizedMap()**
- **java.util.concurrentHashMap** provides a thread-safe hash map
- **CopyOnWriteArrayList** – връща последно актуално копие на итератора. Update-и през това време и да се правят, то при следващо вземане на копие на итератора, ще се е опреснило с промените. Използва се в случаи когато имаме повече четене на елементи, и малко на брой пъти добавяне на елементи.

```
private static void fullLockMap() {  
    final Map<Integer, Integer> numbers = Collections.synchronizedMap(new HashMap<>());  
    synchronized (numbers) {  
        for (int number : numbers.keySet()) {  
            System.out.println(numbers.get(number));  
        }  
    }  
}  
  
private static void concurrentHashMap() {  
    final Map<Integer, Integer> numbers = new ConcurrentHashMap<>();  
    for (int number : numbers.keySet()) {  
        System.out.println(numbers.get(number));  
    }  
}
```

```
private static void fullLock() {
    final List<Integer> numbers = Collections.synchronizedList(new ArrayList<>());
    synchronized (numbers) {
        for (int number : numbers) {
            System.out.println(number);
        }
    }
}

private static void copyOnWrite() {
    final List<Integer> numbers = new CopyOnWriteArrayList<>();
    for (int number : numbers) {
        System.out.println(number);
    }
}
```

Проверка на нишки чрез jps и jstack

jps

```
C:\Users\Martin>jps
6020 XMLServerLauncher
16312 Jps
31596 org.eclipse.equinox.launcher_1.6.400.v20210924-0641.jar
9084 DeadLockExample
```

9084 е id-то на процеса на JVM

jstack като инструмент за разбиране къде точно се случва deadlock например

jstack 9084

```
java.lang.Thread.State: TIMED_WAITING (on object monitor)
    at java.lang.Object.wait(java.base@18.0.2.1/Native Method)
    - waiting on <0x00000007111018a0> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(java.base@18.0.2.1/ReferenceQueue.java:155)
    - locked <0x00000007111018a0> (a java.lang.ref.ReferenceQueue$Lock)
    at jdk.internal.ref.CleanerImpl.run(java.base@18.0.2.1/CleanerImpl.java:140)
    at java.lang.Thread.run(java.base@18.0.2.1/Thread.java:833)
    at jdk.internal.misc.InnocuousThread.run(java.base@18.0.2.1/InnocuousThread.java:10)

Thread-0" #15 prio=5 os_prio=0 cpu=0.00ms elapsed=88.82s tid=0x00000230d8966cc0 nid=22388
[0x000000519eaff000]
    java.lang.Thread.State: BLOCKED (on object monitor)
    at bg.jug.academy.concurrency.DeadLockExample.lambda$0(DeadLockExample.java:18)
    - waiting to lock <0x0000000711104980> (a java.lang.Object)
    - locked <0x0000000711104970> (a java.lang.Object)
    at bg.jug.academy.concurrency.DeadLockExample$$Lambda$1/0x0000000800c009f0.run(Unknown Source)
    at java.lang.Thread.run(java.base@18.0.2.1/Thread.java:833)

Thread-1" #16 prio=5 os_prio=0 cpu=0.00ms elapsed=88.82s tid=0x00000230d89689a0 nid=16460
[0x000000519ebff000]
    java.lang.Thread.State: BLOCKED (on object monitor)
    at bg.jug.academy.concurrency.DeadLockExample.lambda$1(DeadLockExample.java:31)
    - waiting to lock <0x0000000711104970> (a java.lang.Object)
    - locked <0x0000000711104980> (a java.lang.Object)
```

51. Advanced concurrent programming

I. Thread communication

Thread communication

- So far we saw how we can use implicit and explicit locks to provide synchronization between threads
- However locking might not be very flexible in coordinating threads
- A supplement mechanism whereby threads can notify ("wake up") each other and wait to be notified is provided by the JVM
 - Provides a **wait-notify mechanism** between the threads in the JVM, thus creating a **publish-subscribe mechanism** at the thread level in the JVM

Wait and notify

- This is achieved by the **wait**, **notify** and **notifyAll** methods provided by the **java.lang.Object** class
- These methods work over a monitor lock that must be held from threads (i.e. used within a synchronized method or block)
- **notify** wakes up only one thread waiting on the monitor lock while **notifyAll** wakes up all threads
 - Be careful when to use notify and notifyAll - in many cases it is more proper to use notifyAll

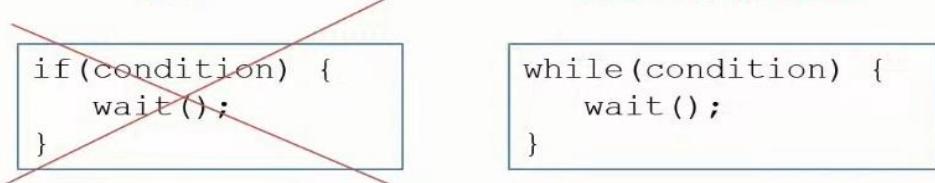
Exiting waits

- Waking up from the **wait** method can happen in the following situations:
 - When **notify/notifyAll** is called from another thread
 - If timeout expires (in case the overloaded **wait** methods are used)
 - The waiting thread is interrupted by calling the **interrupt** method
 - On rare occasions the OS or the JVM may wake up the thread (also called **spurious wakeup**)

Spurious wake-ups

- To guard against spurious wake ups **wait** must always be called in a loop
NO !

CALL WAIT IN A LOOP



Wait/notify example

```
public class PublishSubscribeExample {
    private String message;

    public static void main(String[] args) {
        PublishSubscribeExample example = new PublishSubscribeExample();

        for (int i = 0; i < 100; i++) {
            new Thread(() -> {example.subscribe();}).start();
        }

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); //interrupt flag is being cleared, so if we want the
            layers above to know if the thread was interrupted, we make like this
        }

        new Thread(() -> {example.publish("Hello threads!");}).start();
    }

    public synchronized void publish(String message) {
```

```

        this.message = message;
        System.out.println("Notifying all threads ...");
        notifyAll();
    }

    public synchronized void subscribe(){
        while (message == null){
            try {
                wait();
            } catch (InterruptedException e){
                Thread.currentThread().interrupt();
            }
        }

        System.out.println("Message received: " + message);
    }
}

```

Notifying all threads ...
 Message received: Hello threads!
 Message received: Hello threads!
 Message received: Hello threads!

Conditions

- JDK 5 introduced a more flexible (and preferable way) **to specify wait conditions** using the **java.util.concurrent.lock.Condition** interface
- Additional capabilities of the Condition interface include:
 - `awaitUntil(Date date)` method that waits until a specified date - и ако до това време не се случи, то се събужда нишката
 - **`awaitUninterruptibly()`** method that awaits until the thread is signalled

```

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class PublishSubscribeWithConditionExample {
    private String message;
    private ReentrantLock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public static void main(String[] args) {
        PublishSubscribeWithConditionExample example = new PublishSubscribeWithConditionExample();

        for (int i = 0; i < 100; i++) {
            new Thread(() -> {example.subscribe();}).start();
        }

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); //interrupt flag is being cleared, so if we want the
layers above to know if the thread was interrupted, we make like this
        }

        new Thread(() -> {example.publish("Hello threads!");}).start();
    }

    public synchronized void subscribe() {
        try {
            lock.lock();
            while (this.message == null){
                try {

```

```

        condition.await();
    } catch (InterruptedException e){
        Thread.currentThread().interrupt();
    }
}
System.out.println("Message received: " + this.message);
} finally {
    lock.unlock();
}
}

public synchronized void publish(String message) {
    try {
        lock.lock();
        this.message = message;
        System.out.println("Notifying all threads ...");
        condition.signalAll();
    } finally {
        lock.unlock();
    }
}
}

```

II. Concurrent (general term) collections

- The standard JDK collections (such as LinkedList and ArrayList) are not thread-safe (except for legacy Vector and Hashtable, които **не се препоръчва да се използват** в нови Java приложения)
- The JDK provides several types of thread-safe collections:
 - **Synchronized collections** (such as ones that can be created with the **Collections.synchronizedXXX** methods) - the following methods from Collections class can be used to create synchronized collections:
 - synchronizedCollection
 - synchronizedSet
 - synchronizedSortedSet
 - synchronizedList
 - synchronizedMap
 - synchronizedSortedMap
 - synchronizedNavigableMap
 - **BlockingQueue** - thread-safe lock-based collections provided by the **java.util.concurrent** package such as the implementations of the **BlockingQueue** interface:
 - ArrayBlockingQueue
 - LinkedBlockingQueue
 - LinkedBlockingQueue
 - DelayQueue
 - PriorityBlockingQueue
 - SynchronousQueue
 - Lock-free thread-safe collections (such as ConcurrentHashMap or Copy-on-write) - **не използва locks и synchronized**(синхронизация), а използва **Compare and Swap** техника на база процесорни инструкции, които ни дават възможност да запазваме нещо в паметта само ако дадено друго нещо/стойност вече е запазена в тази памет. **По-бързи тъй като lock/unlock коства ресурси/време.** - before JDK 5 ConcurrentHashMap was NOT lock-free.

1. Synchronized collections

```
private static void syncronizedCollectionExample() {
    List<String> items = new ArrayList<>();
    List<String> synchronizedItems = Collections.synchronizedList(items);
}
```

2. BlockingQueue

- BlockingQueue implementations provide the possibility for threads to wait on operations for adding or removing of elements.
- If the blocking queue is full, threads block until space becomes available for adding an element
- If the blocking queue is empty, threads block until an element is inserted in the queue so it can be removed
- BlockingQueues are used to hold tasks submitted to an Executor thread pool. For example, Fixed thread pool uses by default a LinkingBlockingQueue.

```
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingQueue {
    ArrayBlockingQueue<String> blockingQueue = new ArrayBlockingQueue<String>(50);

    public void add() {
        try {
            blockingQueue.add("first");
            blockingQueue.add("second");
            blockingQueue.add("third");
            System.out.println("Adding items completed");
        }
        catch (Exception e) {
            Thread.currentThread().interrupt();
        }
    }

    public void remove() {
        try {
            System.out.println(blockingQueue.take());
            System.out.println(blockingQueue.take());
            System.out.println(blockingQueue.take());
            System.out.println("Removing items completed");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public static void blockingCollectionExample(){
        BlockingQueue example = new BlockingQueue();
        new Thread(() -> {example.remove();}).start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        new Thread(() -> {example.add();}).start();
    }

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

```
Adding items completed
first
second
third
Removing items completed
```

3. Lock-free collections

- Lock-free collections provided by the JDK come into two flavors:
 - Copy-on-write collections - като добавяме нов елемент, всеки път се прави ново копие, където го записваме този нов елемент. И това изглежда да е скъпа операция.
 - Concurrent collections based on atomic (compare-and-swap) operations

Copy-on-write collections

- Copy-on-write collections create a new collection every time an element is added or removed
- In that regard they are immutable and can be safely accessed from multiple threads
- Copy-on-write collections require extra performance due to the copying of the collection and should be avoided in scenarios where performance is critical
- Copy-on-write collections provided by the JDK include:
 - CopyOnWriteArrayList
 - CopyOnWriteHashSet

```
private static void copyOnWriteExample() {
    CopyOnWriteArrayList<String> copyOnWriteArrayList = new CopyOnWriteArrayList<>();
    copyOnWriteArrayList.add("first");
    copyOnWriteArrayList.add("second");
    copyOnWriteArrayList.add("third");
}
```

Concurrent (specific term) collections

- Concurrent collections do not use locks, but atomic compare-and-swap (CAS) operations
- The JDK provides support for compare-and-swap instructions provided by modern CPUs
- Avoiding locks (thus context switching) makes concurrent collections more performant than synchronized, blocking and copy-on-write in many scenarios
- Concurrent collections provided by the JDK include:
 - ConcurrentHashMap
 - ConcurrentLinkedQueue
 - ConcurrentLinkedDeque
 - ConcurrentSkipListMap
 - ConcurrentSkipListSet

ConcurrentHashMap

- ConcurrentHashMap provides additional thread-safe atomic operations over a traditional HashMap such as:
 - getOrDefault(key, value)
 - putIfAbsent(key, value)
 - remove(key, value)
 - replace(key, oldValue, newValue)
 - replaceAll(function)
 - computeIfAbsent(key, function)
 - computeIfPresent(key, function)
 - compute(key, function)

- merge(key, value, function)

```
//Ако използваме обикновен HashMap, не винаги ще ни връща 100 в долния пример със 100 нишки!!!
public class Main {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<Integer, Integer> map = new ConcurrentHashMap<>();
        map.put(1, 0);
        for (int i = 0; i < 100; i++) {
            new Thread(() -> {
                map.compute(1, (key, value) -> {
                    return value + 1;
                });
            }).start();
        }

        Thread.sleep(1000);
        System.out.println(map.getOrDefault(1, -1));
    }
}
```

III. Synchronizers

- Synchronizers provide more specific mechanisms for synchronization between a number of threads
- The JDK provides several synchronizers that can be used by applications:
 - CountDownLatch
 - CyclicBarrier
 - Semaphore
 - Exchanger
 - Phaser

Java concurrent animated - визуално описани

<https://github.com/vgrazi/java-concurrent-animated>

```
/java-concurrent-animated$ mvn package -DskipTests
/java-concurrent-animated/target$ java -jar javaConcurrentAnimated.jar
```

CountDownLatch

- Used when a predefined number of releases should happen before a thread is awakened
- Един път казваме да изчака, и намаляме с countDown()

```
public class CountDownLatch {
    public static void main(String[] args) {
        java.util.concurrent.CountDownLatch latch = new java.util.concurrent.CountDownLatch(5);
        new Thread(() -> {
            try {
                latch.await();
                System.out.println("Workers have finished !");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();

        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                System.out.println("Starting worker: " + Thread.currentThread().getName());
                latch.countDown();
            }).start();
        }
    }
}
```

```
Starting worker: Thread-1
Starting worker: Thread-2
Starting worker: Thread-4
Starting worker: Thread-3
Starting worker: Thread-5
Workers have finished !
```

```
Ако CountDownLatch(3), тогава резултатът е:
Starting worker: Thread-1
Starting worker: Thread-5
Starting worker: Thread-4
Workers have finished !
Starting worker: Thread-3
Starting worker: Thread-2
```

CyclicBarrier

- Allows a number of threads to await for a predefined number of waits after which they are released
- Извикваме n на брой пъти преди да се освободи изпълнението на нишките

```
public class CyclicBarrier {  
  
    public static void main(String[] args) {  
        java.util.concurrent.CyclicBarrier barrier = new java.util.concurrent.CyclicBarrier(3, () ->  
        {  
            System.out.println("Workers have finished!");  
        });  
  
        for (int i = 0; i < 5; i++) {  
            new Thread(() -> {  
                System.out.println("Starting worker: " + Thread.currentThread().getName());  
                try {  
                    barrier.await();  
                } catch (InterruptedException e) {  
                    Thread.currentThread().interrupt();  
                } catch (BrokenBarrierException e) {  
                    System.out.println("Ending worker: " + Thread.currentThread().getName());  
                }  
            }).start();  
        } //barrier can be reset with barrier.reset()  
    }  
}
```

```
Starting worker: Thread-2
Starting worker: Thread-0
Starting worker: Thread-1
Workers have finished!
Starting worker: Thread-3
Starting worker: Thread-4
```

Semaphore

- Each thread is blocked until a permit is available, semaphore is initialized with a number of permits
- Когато искаме в даден или във всеки момент от време да се изпълняват едновременно определен брой нишки

```
public class Semaphore {
```

```

public static void main(String[] args) throws InterruptedException {
    java.util.concurrent.Semaphore semaphore = new java.util.concurrent.Semaphore(3);
    for (int i = 0; i < 5; i++) {
        new Thread(() -> {
            try {
                System.out.println("Starting worker: " + Thread.currentThread().getName());

                //Acquires a permit from this semaphore, blocking until one is available, or the
                thread is interrupted.
                //Acquires a permit, if one is available and returns immediately, reducing the
                number of available permits by one.
                semaphore.acquire();

                System.out.println("Ending worker: " + Thread.currentThread().getName());
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();
    }

    Thread.sleep(2000);
    semaphore.release(); //Първите 3 нишки заемат трите части на семафората
    semaphore.release(); //четвърта и пета нишка се блокират докато не се освободи слот от
    семафората за тях
}
}

```

Starting worker: Thread-2
 Starting worker: Thread-0
 Ending worker: Thread-2
Starting worker: Thread-3
 Ending worker: Thread-3
 Starting worker: Thread-1
 Starting worker: Thread-4
 Ending worker: Thread-0
 Ending worker: Thread-1
 Ending worker: Thread-4

Exchanger

- Provides the possibility to two threads to exchange (swap) objects

```

public class Exchanger {

    public static void main(String[] args) {
        java.util.concurrent.Exchanger exchanger = new java.util.concurrent.Exchanger();

        for (int i = 0; i < 2; i++) {
            new Thread(() -> {
                try{
                    Random random = new Random();
                    Integer value = random.nextInt();
                    Integer exchanged = (Integer) exchanger.exchange(value);
                    System.out.println("Exchanged value " + value + " for " + exchanged);
                } catch (InterruptedException e){
                    Thread.currentThread().interrupt();
                }
            }).start();
        }
    }
}

```

Веднъж се разменят стойностите, като първата нишка блокира, докато не е готова и втората- така че да обменят стойности.

Exchanged value 1878992229 for 2140338631

Exchanged value 2140338631 for 1878992229

Phaser

- Similar to CountDownLatch and CyclicBarrier, but provides the ability **to change dynamically the number of awaiting threads** before they can proceed

```
public class Phaser {  
  
    public static void main(String[] args) {  
        java.util.concurrent.Phaser phaser = new java.util.concurrent.Phaser(1);  
        phaser.bulkRegister(2); //още два пъти да се извика метода, преди да продължи приложението  
  
        for (int i = 0; i < 3; i++) {  
            new Thread(() -> {  
                System.out.println(Thread.currentThread().getName() + " arriving at phaser");  
                phaser.arriveAndAwaitAdvance();  
                System.out.println(Thread.currentThread().getName() + " leaving the phaser");  
            }).start();  
        }  
    }  
}
```

Thread-0 arriving at phaser

Thread-2 arriving at phaser

Thread-1 arriving at phaser

Thread-0 leaving the phaser

Thread-2 leaving the phaser

Thread-1 leaving the phaser

АКО `phaser.bulkRegister(3)`, то чакаме за общо четири нишки да влязат, а в нашия случай четвърта никога не идва....

Thread-1 arriving at phaser

Thread-0 arriving at phaser

Thread-2 arriving at phaser

IV. Atomic operations

Compare-and-Swap (CAS)

- Compare-and-Swap is an atomic instruction that compares the location in memory with a given value, and only if they are equals then sets a new value
- CAS is used to implement atomic operations that achieve **optimistic locking** and are thread-safe

```
// represented by the following pseudocode  
if (memoryLocation != value) {  
    return false;  
}  
memoryLocation = newValue;  
return true;
```

- The JDK provides support for calling CAS instructions through native code as provided by the **jdk.internal.misc.Unsafe** class.
 - Това е клас, който се ползва вътрешно от самата JVM на много места при операции свързани със синхронизация на нишки; Преди се е използвал този клас за приложения, които искат да заделят отделна памет извън HEAP паметта на JVM (unmanaged memory not allocated for the JVM); Не случайно се казва Unsafe - защото ако не се използва внимателно, може да доведе до сериозни проблеми
 - Apart from performing CAS operations the Unsafe class also provides the possibility to access off-heap memory. But since it is an internal class, it is encapsulated as of JDK 9 so cannot be used directly
- An alternative of Unsafe as of JDK 9 that provides support for CAS is provided by the **java.lang.invoke.VarHandle** class

```
import jdk.internal.misc.Unsafe;
import java.lang.invoke.VarHandle;

public class Main {
    public static void main(String[] args) {
        Unsafe.compareAndSetInt();
        VarHandle.compareAndSet();
    }
}
```

Atomic variables

- Maintaining a single variable that is updatable from many threads is a common scalability issue
- Atomic variables are already present in the JDK - they serve as a means to implement updatable variables in a multithreaded environment
- Atomic variables are part of the **java.util.concurrent.atomic** package
- They make use of the CAS support provided by the JDK to provide performant thread-safe operations over shared variables
- The other utilities that we already discussed that make use of CAS are concurrent collections (specific term) like **ConcurrentHashMap**

Atomic variables are provided by the following classes:

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicLong
- AtomicLongArray
- AtomicReference
- AtomicReferenceArray
- DoubleAccumulator
- DoubleAdder
- LongAccumulator
- LongAdder

Пример за AtomicInteger

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

    AtomicInteger value = new AtomicInteger();
    Thread thread = new Thread(() -> {
        value.getAndIncrement();
    });
}
```

```

        thread.start();
        value.getAndAdd(10);
        try {
            thread.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(value.get()); //11
    }

java.util.concurrent.atomic.AtomicInteger:
private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe();

public final int getAndIncrement() {
    return U.getAndAddInt(this, VALUE, 1);
}

jdk.internal.misc.Unsafe:
@HotSpotIntrinsicCandidate
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}

@HotSpotIntrinsicCandidate
public final boolean weakCompareAndSetInt(Object o,
                                             long offset, //offset в паметта
                                             int expected, //ако тази стойност вече е налична в паметта
                                             int x) { //новата стойност x ако expected е налично
    return compareAndSetInt(o, offset, expected, x);
}

/**
 * Atomically updates Java variable to {@code x} if it is currently
 * holding {@code expected}.
 *
 * <p>This operation has memory semantics of a {@code volatile} read
 * and write. Corresponds to C11 atomic_compare_exchange_strong.
 *
 * @return {@code true} if successful
 */
@HotSpotIntrinsicCandidate
public final native boolean compareAndSetInt(Object o, long offset,
                                              int expected,
                                              int x);

```

native ще рече, че се извиква отдолу в JVM, вероятно на C++, Compare-and-Swap процесорна инструкция.

Пример за DoubleAccumulator

```

import java.util.concurrent.atomic.DoubleAccumulator;

public class Main {
    public static void main(String[] args) {

        DoubleAccumulator accumulator = new DoubleAccumulator((x, y) -> x + y, 0);
        Thread thread = new Thread(() -> {
            accumulator.accumulate(0.9);

```

```

    });

    thread.start();
    accumulator.accumulate(10.1);
    try {
        thread.join();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    System.out.println(accumulator.get()); //11
}
}

```

Пример за AtomicReference

Използва се за обекти от всякахъв тип, включително и за Стинг.

```

public class Main {
    public static void main(String[] args) {
        AtomicReference<String> reference = new AtomicReference<String>("first");
        Thread thread = new Thread(() -> {
            reference.getAndAccumulate(" some text", (x, y) -> x + y);
        });

        thread.start();
        reference.getAndAccumulate(" otherText", (x, y) -> x + y);
        try {
            thread.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(reference.get());
    }
}

```

Не се гарантира поредността, но се гарантира thread safety
first otherText some text

StringBuffer

Thread-safe operations.

Но не е реализирано със CAS, а със **copy-on-write** - т.е. за повече обем данни може да е бавна операцията

V. Concurrency utilities

Fork/join framework

- **ForkJoinPool** is an implementation of the **ExecutorService** interface introduced in JDK 7
- Provides the possibility to execute tasks that can be organized in a **divide-and-conquer** manner - разбиване на подзадачи например като сортиране с **mergeSort** алгоритъм (see Algorithms with Java notes) при многонишкова среда
- Tasks submitted to the **ForkJoinPool** are represented by a **ForkJoinTask** instance
- Typical implementations of parallel tasks do not extend directly **ForkJoinTask**
- **RecursiveTask** instances can be used to execute parallel tasks that return a result

- **RecursiveAction** instances can be used to execute parallel tasks that do NOT return a result
- A **global ForkJoinPool instance** is used for any ForkJoinTasks that are not submitted to a particular ForkJoinPool
- To get a reference to the global ForkJoinPool instance, the static **ForkJoinPool.commonPool()** method can be used
- This is the case with **parallel streams**: they make use of the common ForkJoinPool for task execution

```

import java.util.concurrent.RecursiveAction;

public class NumberFormatAction extends RecursiveAction {
    int start;
    int end;

    public NumberFormatAction(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected void compute() {
        if (end - start <= 2) {
            System.out.println(start + " " + end);
        } else {
            int mid = start + (end - start) / 2;
            NumberFormatAction left = new NumberFormatAction(start, mid);
            NumberFormatAction right = new NumberFormatAction(mid + 1, end);
            java.util.concurrent.ForkJoinTask.invokeAll(left, right);
        }
    }
}

public static void main(String[] args) {
    NumberFormatAction action = new NumberFormatAction(1, 50);
    ForkJoinPool.commonPool().invoke(action);
}

```

Parallel streams / parallel programming

- Parallel streams can be created as regular streams with the **parallelStream** method

```

List list = new ArrayList();
list.parallelStream();

```

- Parallel streams should be used **only for time-intensive tasks rather than IO**
- In many cases regular streams outperform parallel ones so they need to be used with caution - затова винаги е добре когато мислим да използваме паралелни потоци, то да им сравним бързодействието с нормални потоци

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;

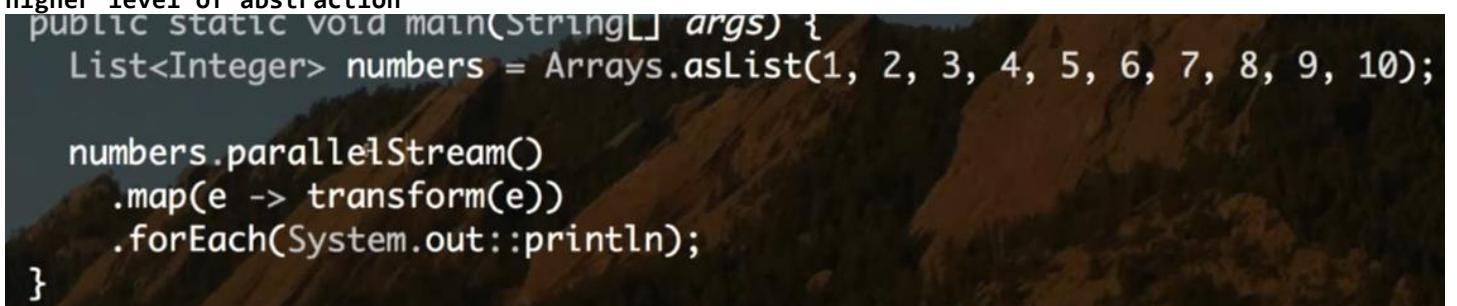
    while (true){
        z++;
    }
}
}

```

`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-a

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 ????

$$\#T \leq \frac{\# \text{ of Cores}}{1 - \text{blocking factor}}$$

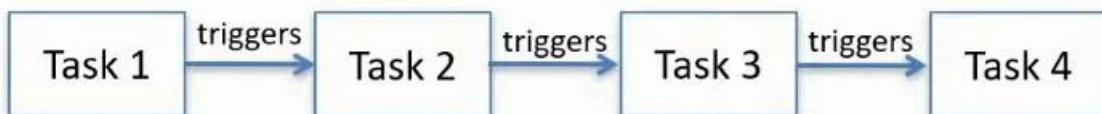
$$0 \leq \text{blocking factor} < 1$$

Configuring number of threads JVM wide:

Djava.util.concurrent.ForkJoinPool.common.parallelism=100 100 нитки

CompletableFuture

- Provides a facility to create a chain of dependant non-blocking tasks
- **No divide-and-conquer** - it could be any relation of operations - i.e. a smarter way of organization when using CompletableFuture
- An asynchronous task can be triggered as the result of a completion of another task



- A CompletableFuture may be completed/canceled by a thread prematurely (преждевременно)
- Provides a very flexible API that allows additionally:
 - To combine the result of multiple tasks in a CompletableFuture
 - To provide synchronous/asynchronous callbacks upon completion of a task
 - To provide a CompletableFuture that executes when first task in a group completes
 - To provide a CompletableFuture that executes when all tasks in a group complete

1. Най-простиат пример

```
import java.util.concurrent.CompletableFuture;

public class Main {
    public static void main(String[] args) {
        CompletableFuture<Integer> task1 = new CompletableFuture<>();
```

```

    // forcing completing of future by specifying result
    boolean complete = task1.complete(10);
}
}

2.

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Main {
    public static void main(String[] args) throws ExecutionException, InterruptedException

        CompletableFuture<Integer> task1 = CompletableFuture.supplyAsync(() -> {
            //some code logic here
            return 10;
        });

        //executed on completion of the future
        CompletableFuture<Integer> integerCompletableFuture = task1.thenApply((x) -> x * x);
        System.out.println(integerCompletableFuture.get()); //100

        //executed in case of exception or completion of the future
        task1.handle((x, y) -> {
            return .....
        });
        System.err.println(task1.get());

        //can be completed prematurely with a result
        task1.complete(20);
    }
}

```

3. Верига от задачи

```

CompletableFuture<Object> prev = null;
Supplier<Object> supplier = () -> {};

for (int i = 0; i < count; i++) {
    CompletableFuture<Object> task;
    if (prev != null) {
        task = prev.thenCompose(x -> {
            return CompletableFuture.supplyAsync(supplier);
        });
    } else {
        task = CompletableFuture.supplyAsync(supplier);
    }

    prev = task;
}

prev.get(); //изпълнява целия chain

```

A few more things

- There is a **Timer** utility/class that can be used for scheduling tasks, but a **scheduled executor thread pool is more preferable as a more robust alternative**
- A **ThreadLocalRandom** utility is provided since JDK 7 that can be used to generate random numbers for the current thread

- A **Flow** interface introduced in JDK 9 provides a **reactive streams specification for the JDK** that can be used to provide a publish-subscribe mechanism - frameworks like in Spring (implemented by WebFlux reactor) and in Quarkus(implemented by Mutiny)
 - Парадигмата за реактивно програмиране идва от Microsoft като постепенно се имплементира в различни програмни езици. **Идеята на реактивното програмиране е да се изпълняват само неблокиращи операции и се касае за IO операции**(в това число http протокол, web socket, и т.н.) в не малка част от случаите! Идеята за реактивното програмиране решава въпроса за скалируемост - да може например повече да се бомбардирва със http заявки даден http endpoint

VI. Testing concurrent applications

- Testing concurrent applications for correctness is inherently difficult
- In many cases non-deterministic unit tests are written that try to simulate running a particular piece of code in a multithreaded manner
- Testing frameworks provide certain utilities that can be used to facilitate testing of concurrent applications
- A classical way of running multiple threads against code-under-test in a unit test is to use **CountDownLatch**:

```
@RepeatedTest(10)
public void testLinkedList()
    throws InterruptedException {
    CountDownLatch latch = new CountDownLatch(100);
    for(int i = 0; i < 100; i++) {
        new Thread( () -> {
            // unit under test ...
            latch.countDown();
        }).start();
    }
    latch.await();
    // perform asserts
}
```

- Frameworks like **ThreadWeaver** provide the possibility to interleave execution threads - interleaving happens using breakpoints (line by line) so that the unit-under-test is tested using different thread ordering
- For performance testing a framework like **JMH**(the **Java Microbenchmark Harness**) can be used to perform proper application warmup before the test is executed - самото JDK се тества с JMH

VII. Latest concurrency enhancements

- At the time of 2023/2024, JDK 21 is the latest LTS (long-term support) release of JDK 21
- A few but MAJOR additions for writing concurrent applications are introduced:
 - Virtual threads (aka fibers) provided by **Project Loom**

- Scoped values
- Structured concurrency

Virtual threads

- Virtual threads are “lightweight” threads scheduled by the JVM
- In comparison with standard platform threads, they are not bound directly to OS threads
- They are still executed by an OS thread
- They are an instance of the **java.lang.Thread** class - същата инстанция както и стандартните нишки
- Virtual threads are very suitable for high-throughput concurrent application
- Typical example are Java server applications where many threads are blocked and waiting
- Not suitable for long running operations

```

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {
    public static void main(String[] args) {

    }

    private static void createSimpleVirtualThread() throws InterruptedException {
        Thread thread = Thread.ofVirtual().start(() -> {...});
        thread.join();
    }

    private static void createVirtualThreadPool() throws InterruptedException, ExecutionException {
        ExecutorService virtualThreadPool = Executors.newVirtualThreadPerTaskExecutor();
        Future<?> task = virtualThreadPool.submit(() -> {...});
        task.get();
    }
}

```

52. Asynchronous programming

What is Asynchronous Programming?

Let's delve into an analogy of asynchronous programming: Imagine waking up in the morning and starting to boil some eggs for breakfast. Instead of waiting idly for them to cook, you begin another task, such as baking bacon and toast. In this scenario, you're akin to a thread executing I/O operations - you initiate them and let them run independently. Both tasks are non-blocking for you; while the food cooks, you're free to sit at the table, enjoy some YouTube videos, or even take a shower. Once any task is completed (for instance, the bacon is ready), you handle it by placing it on your plate. Similarly, when the eggs are done, you remove them from the water and prepare them to be eaten.

Consider another real-world example of asynchronous operations: a waiter at a restaurant taking orders. Upon receiving an order from a customer, he relays it to the kitchen for preparation. Rather than waiting beside the kitchen counter for the dish to be ready – an approach similar to a Java application operating on a single thread - the waiter moves on to attend to another customer's order. As soon as a meal is prepared, he retrieves it from the kitchen and serves it to the respective customer.

This process mirrors how an application thread behaves during an API call, initiating a time-consuming operation on an external system or executing a complex database query that takes several seconds or more.

Real-life scenarios also require exception handling, akin to software development. For instance, if the waiter takes an order for pasta bolognese but finds out that the kitchen has run out of beef, it poses a resource synchronisation issue typical in asynchronous operations.

Modern web applications, particularly those hosted in cloud environments, need to accommodate thousands of simultaneous users. This scalability is achieved not only through service replication, such as pod replication in EKS, but also by making efficient use of threads within each pod at the application level. Asynchronous operations facilitate non-blocking I/O, leading to more efficient use of resources.

Async vs parallel operations: Asynchronous programming focuses on non-blocking tasks, while parallel programming involves executing multiple computations simultaneously, leveraging multi-core processors. Both approaches improve performance but are used for different purposes.

Async Programming in Other Languages

C# in the .NET framework utilises the `Task<T>` class and `async` and `await` keywords, making asynchronous programming more straightforward and cleaner. An `async` method returns a `Task` or `Task<T>`, which represents ongoing work. C# similar to Java utilises a thread pool for executing asynchronous tasks. When an `async` method awaits an asynchronous operation, the current thread is returned to the thread pool until the awaited operation completes.

C/C++

```
public async Task<string> GetDataAsync() {
    var data = await GetDataFromDatabaseAsync();
    return data;
}
```

JavaScript handles asynchronous operations through `Promises` and `async/await` syntax, fitting its event-driven nature. A `Promise` represents an operation that hasn't been completed yet but is expected in the future. JavaScript, particularly in the `Node.js` environment, operates on a single-threaded event loop for handling asynchronous operations.

JavaScript

```
async function fetchData() {
    let response = await fetch('https://api.example.com/data');
    let data = await response.json();
    console.log(data);
}
```

In JAVA it is `CompletableFuture`

In JS it is `Promises`

Java's CompletableFuture Class

Java's `CompletableFuture` class was introduced in **Java 8**. `CompletableFuture` is part of Java's `java.util.concurrent` package and provides a way to write asynchronous code by representing a future result that will eventually appear. It lets us perform operations like calculation, transformation, and action on the result without blocking the main thread. This approach helps in writing non-blocking code where the computation can be completed by a different thread at a later time.

`CompletableFuture` and the broader **Java Concurrency API** make use of thread pools (**like the ForkJoinPool**) for executing asynchronous operations. This allows Java applications to handle multiple asynchronous tasks efficiently by leveraging multiple threads.

```
CompletableFuture.supplyAsync(() -> {
    return "Result of the asynchronous computation";
})
```

```

.thenAccept(System.out::println);

public static CompletableFuture<Integer> create() {
    return CompletableFuture.supplyAsync(() -> 2);
}

```

In Java, when a **CompletableFuture** operation is waiting on a dependent future or an asynchronous computation, it doesn't block the waiting thread. Instead, the completion of the operation triggers the execution of dependent stages in the **CompletableFuture** chain, potentially on a different thread from the thread pool.

Example Scenario with CompletableFuture

Let's consider a scenario where we need to perform a series of dependent and independent asynchronous operations:

1. **Fetch User Details**: Given a userID, we first retrieve the user's details asynchronously.
2. **Fetch Credit Score**: Once we have the user's details, we fetch their credit score.
3. **Calculate Account Balance**: Independently, we also calculate the user's account balance from a different source.
4. **Make a Decision**: Finally, we combine the credit score and account balance to make a financial decision.
5. **Handle Potential Errors**

Step 1: Fetching User Details Asynchronously

We start by simulating an asynchronous operation to fetch user details using **supplyAsync**. This returns a **CompletableFuture** that will complete with the user details:

```

CompletableFuture<String> getUserDetailsAsync(String userId) {
    return CompletableFuture.supplyAsync(() -> "UserDetails for " + userId);
}

```

Step 2: Transforming and Fetching Credit Score

Next, we use **thenApply** to transform the result (e.g., formatting user details) and **thenCompose** to fetch the credit score, demonstrating the chaining of asynchronous operations:

```

CompletableFuture<String> userDetailsFuture = getUserDetailsAsync("userId123")
    .thenApply(userDetails -> "Transformed " + userDetails);

```

```

CompletableFuture<Integer> creditScoreFuture = userDetailsFuture
    .thenCompose(userDetails -> getCreditScoreAsync(userDetails));

```

thenApply is for synchronous transformations, while **thenCompose** allows for chaining another asynchronous operation that returns a **CompletableFuture**.

Step 3: Calculating Account Balance in Parallel

We calculate the account balance using another asynchronous operation, showcasing how independent futures can run in parallel:

```

CompletableFuture<Double> accountBalanceFuture = calculateAccountBalanceAsync("userId123");

```

Step 4: Combining Results and Making a Decision

With **thenCombine** we merge the results of two independent **CompletableFuture** – credit score and account balance – to make a decision:

```

CompletableFuture<Void> decisionFuture = creditScoreFuture
    .thenCombine(accountBalanceFuture, (creditScore, accountBalance) ->
        makeDecisionBasedOnCreditAndBalance(creditScore, accountBalance))
    .thenAccept(decision -> System.out.println("Decision: " + decision));

```

Step 5. Error Handling

Error handling is crucial in asynchronous programming. We use `exceptionally` to handle any exceptions that may occur during the asynchronous computations, providing a way to recover or log errors:

```

.exceptionally(ex -> {
    System.err.println("An error occurred: " + ex.getMessage());
    return null;
});

```

CompletableFuture from Indian UK guy

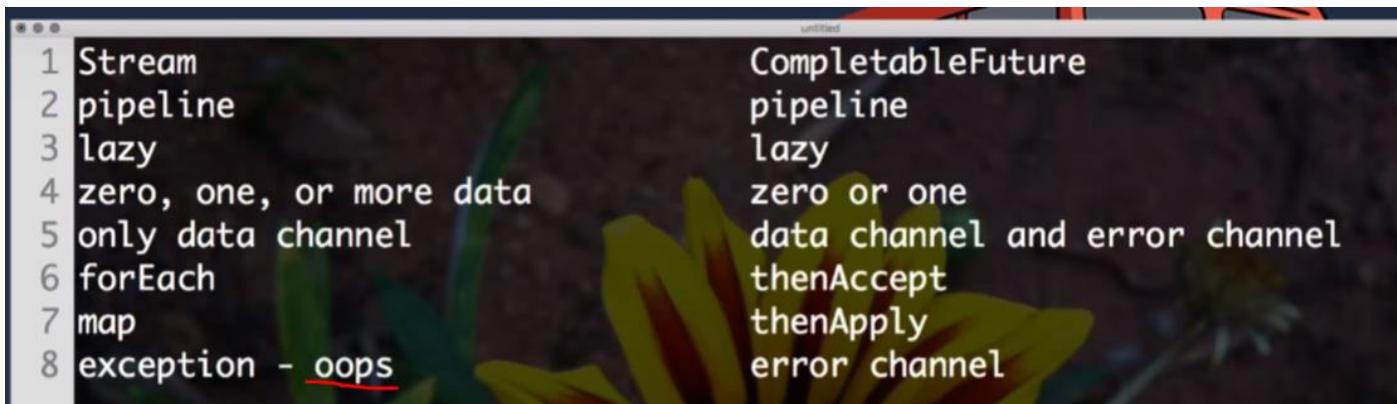
Some Info

```

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"));

```



```

.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 CompletableFuture<Integer> create() {
    return CompletableFuture.supplyAsync(() -> 2);
}

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 нишо не сме свършили

Състояния на CompletableFuture:

Pending (final)
Resolved state (final)
Rejected (final)

Future.orTimeout();

Combine and compose

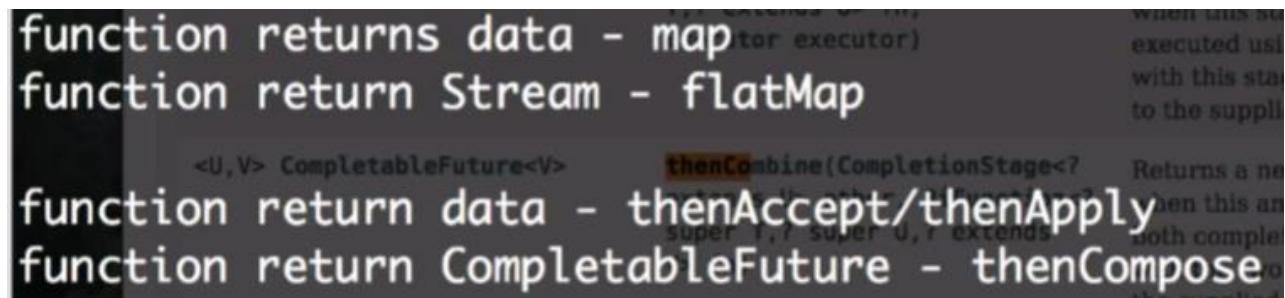
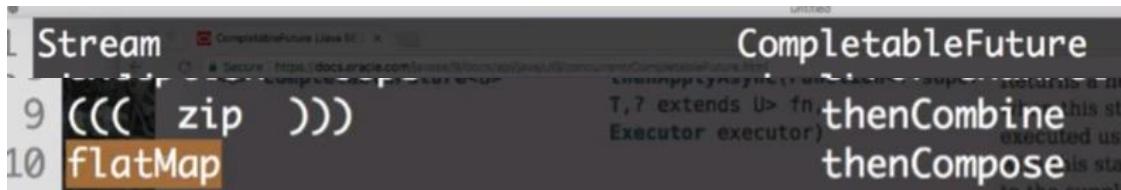
//JavaScript
then(e => func(e)) ➔

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) {  
    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));

```

Using CompletableFutures Effectively

Most Important Methods

1. supplyAsync

The `supplyAsync` method is part of the `CompletableFuture` class introduced in Java 8, residing in the `java.util.concurrent package`. It's designed to run a piece of code asynchronously and return a `CompletableFuture` that will be completed with the `value` obtained from that code. Essentially, it allows you to execute a `Supplier<T>` asynchronously, where `T` is the type of `value` returned by the `Supplier`.

Syntax and Usage:

```

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
    - U: The type of value obtained from the Supplier<U>
    - supplier: A Supplier<U> that provides the value to be used in the completion of the
returned CompletableFuture<U>

```

Simple example:

```

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Simulate a long-running operation
    try {
        TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }

    return "Result of the asynchronous operation";
});

```

When you invoke `supplyAsync`, it executes the given `Supplier` asynchronously (usually in a different thread). The method immediately returns a `CompletableFuture` object. This `CompletableFuture` will be completed in the future when the `Supplier` finishes its execution, with the result being the value provided by the `Supplier`.

It allows the main thread to continue its operations without waiting for the task to complete. This is particularly useful in web applications or any I/O-bound applications where you don't want to block the current thread.

By default, tasks submitted via `supplyAsync` without specifying an executor are executed in the common fork-join pool (`ForkJoinPool.commonPool()`). However, you can also specify a custom `Executor` if you need more control over the execution environment:

```

Executor executor = Executors.newCachedThreadPool();
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Task here
    return "Result";
}, executor);

```

2. runAsync

`CompletableFuture.runAsync` is akin to `CompletableFuture.supplyAsync` but for scenarios where you don't need to return a value from the asynchronous operation. Both methods are intended for executing tasks asynchronously, but they differ in their return types and the type of tasks they're suited for.

`runAsync` is used to execute a `Runnable` task asynchronously, which does not return a result. Since `Runnable` does not produce a return value, `runAsync` returns a `CompletableFuture<Void>`.

```
static CompletableFuture<Void> runAsync(Runnable runnable)
static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
```

- **Asynchronous Execution:** Executes the given `Runnable` task in a separate thread, allowing the calling thread to proceed without waiting for the task to complete.
- **No Return Value:** Suitable for asynchronous tasks that perform actions without needing to return a result, such as logging, sending notifications, or other side effects.
- **Custom Executor Support:** Allows specifying a custom `Executor` for more control over task execution, such as using a dedicated thread pool.

Here's an example that demonstrates using `runAsync` to execute a simple asynchronous task:

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
    // Simulate a task that takes time but doesn't return a result
    try {
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Task completed");
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
});

// Do something else while the task executes

future.join(); // Wait for the task to complete if necessary
```

3. get()

The `get()` method blocks the current thread until the `CompletableFuture` completes, either normally or exceptionally. Once the future completes, `get()` returns the result of the computation if it completed normally, or throws an exception if the computation completed exceptionally.

1. **Blocking Behaviour:** Like `join()`, `get()` is a blocking call. It makes the caller thread wait until the `CompletableFuture`'s task is completed.

2. **Checked Exceptions:** `get()` can throw checked exceptions, including:

a. `InterruptedException`: If the current thread was interrupted while waiting.

b. `ExecutionException`: If the computation threw an exception. This exception wraps the actual exception thrown by the computation, which can be obtained by calling `getCause()` on the `ExecutionException`.

3. **Timeout:** The overloaded version of `get(long timeout, TimeUnit unit)` allows specifying a maximum wait time. If the timeout is reached before the future completes, it throws a `TimeoutException`, providing a mechanism to avoid indefinite blocking.

4. **Use Case:** Use `get()` when you need to handle checked exceptions explicitly, or when you need to retrieve the result of the computation within a certain timeframe.

Example:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Simulate a long-running operation
    try {
```

```

        TimeUnit.SECONDS.sleep(2); // 2-second delay
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return "Result of the asynchronous operation";
});

try {
    // Attempt to retrieve the result, waiting up to 3 seconds
    String result = future.get(3, TimeUnit.SECONDS);
    System.out.println(result);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    System.out.println("The current thread was interrupted while waiting.");
} catch (ExecutionException e) {
    System.out.println("The computation threw an exception: " + e.getCause());
} catch (TimeoutException e) {
    System.out.println("Timeout reached before the future completed.");
}

```

4. join()

The `join` method on a `CompletableFuture` is a blocking call that causes the current thread to wait until the `CompletableFuture` is completed. During this waiting period, the current thread is inactive, essentially "joining" the completion of the task represented by the `CompletableFuture`.

- **Blocking Behaviour:** `join()` blocks until the `CompletableFuture` upon which it is called completes, either normally or exceptionally. It makes the asynchronous operation synchronous for the calling thread, as the thread will not proceed until the future is completed.
- **Exception Handling:** Unlike `get()`, which throws checked exceptions (such as `InterruptedException` and `ExecutionException`), `join()` is designed to throw an unchecked exception (`CompletionException`) if the `CompletableFuture` completes exceptionally. This can simplify error handling in certain contexts where checked exceptions are undesirable.
- **Usage:** It's typically used when you need to synchronize asynchronous computation at some point, for example, when the result of the asynchronous computation is required for subsequent operations, or at the end of a program to ensure that all asynchronous tasks have completed.

Example scenario:

If you have a main application thread that kicks off an asynchronous task using `CompletableFuture.runAsync()` or `CompletableFuture.supplyAsync()`, and later in the program you need the result of that task or need to ensure that the task has completed before proceeding, you might call `join()`:

```

CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
    // Long-running task
    try {
        TimeUnit.SECONDS.sleep(1); // Simulates a delay
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    System.out.println("Async task finished");
});

System.out.println("Waiting for the async task to complete...");
future.join(); // Blocks here until the above task completes
System.out.println("Main thread can now proceed");

```

5. thenApply(Function<? super T, ? extends U> fn)

- **Purpose:** Applies a synchronous transformation function to the result of the **CompletableFuture** when it completes.
- **Behaviour:** Executes on the same thread that completed the previous stage, or in the thread that calls `get()` or `join()` if the future has already completed.
- **Return Type:** `CompletableFuture<U>` where U is the type returned by the function.

6. thenApplyAsync(Function<? super T, ? extends U> fn)

- **Purpose:** Similar to `thenApply`, but the transformation function is executed asynchronously, typically using the default executor.
- **Behaviour:** Can execute the function in a different thread, providing better responsiveness and throughput for tasks that are CPU-intensive or involve blocking.
- **Return Type:** `CompletableFuture<U>`

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 42);
CompletableFuture<String> applied = future.thenApply(result -> "Result: " + result);
applied.thenAccept(System.out::println); // Prints: Result: 42
CompletableFuture<String> appliedAsync = future.thenApplyAsync(result -> "Async Result: " + result);
appliedAsync.thenAccept(System.out::println); // Prints: Async Result: 42
```

7. thenCombine(CompletionStage<? extends V> other, BiFunction<? super T, ? super V, ? extends U> fn)

- **Purpose:** Combines the result of this **CompletableFuture** with another asynchronously computed value. The combination is done when both futures complete.
- **Behaviour:** The **BiFunction** provided is executed synchronously, using the thread that completes the second future.
- **Return Type:** `CompletableFuture<U>`

8. thenCombineAsync(CompletionStage<? extends V> other, BiFunction<? super T, ? super V, ? extends U> fn)

- **Purpose:** Similar to `thenCombine`, but the **BiFunction** is executed asynchronously.
- **Behaviour:** Useful when the combination function is computationally expensive or involves blocking.
- **Return Type:** `CompletableFuture<U>`

```
CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> 40);
CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> 2);
CompletableFuture<Integer> combined = future1.thenCombine(future2, Integer::sum);
combined.thenAccept(result -> System.out.println("Sum: " + result)); // Prints: Sum: 42
```

```
CompletableFuture<Integer> combinedAsync = future1.thenCombineAsync(future2, Integer::sum);
combinedAsync.thenAccept(result -> System.out.println("Async Sum: " + result)); // Prints: Async Sum: 42
```

9. thenAccept and thenAcceptAsync

- **Purpose:** Consumes the result of the **CompletableFuture** without returning a result. `thenAccept` is synchronous, while `thenAcceptAsync` is asynchronous.
- **Use Case:** Useful for executing side-effects, such as logging or updating a user interface, with the result of the **CompletableFuture**.

```
CompletableFuture<Void> accepted = CompletableFuture.supplyAsync(() -> "Hello")
.thenAccept(result -> System.out.println(result + ", World!")); // Prints: Hello, World!
```

```
CompletableFuture<Void> acceptedAsync = CompletableFuture.supplyAsync(() -> "Async Hello")
```

```
.thenAcceptAsync(result -> System.out.println(result + ", World!")); // Prints: Async Hello, World!
```

10. thenRun and thenRunAsync

- **Purpose:** Executes a `Runnable` action when the `CompletableFuture` completes, without using the result of the future. `thenRun` is synchronous, while `thenRunAsync` is asynchronous.
- **Use Case:** Useful for triggering actions that do not depend on the future's result, such as signalling completion or cleaning up resources.

```
CompletableFuture.supplyAsync(() -> "Task completed")
    .thenRun(() -> System.out.println("Running next step...")); // Executed after the supplyAsync task
```

```
CompletableFuture.supplyAsync(() -> "Async task completed")
    .thenRunAsync(() -> System.out.println("Running async next step...")); // Executed asynchronously after the supplyAsync task
```

11. exceptionally(Function<Throwable, ? extends T> fn)

- **Purpose:** Handles exceptions arising from the `CompletableFuture` computation, allowing for a fallback value to be provided or a new exception to be thrown.
- **Use Case:** Essential for robust error handling in asynchronous programming, allowing for recovery or logging of failures.

```
CompletableFuture<String> exceptionFuture = CompletableFuture.supplyAsync(() ->
{
    if (true) throw new RuntimeException("Exception!");
    return "No Exception";
})
.exceptionally(ex -> "Exception Handled: " + ex.getMessage());

exceptionFuture.thenAccept(System.out::println); // Prints: Exception Handled: java.lang.RuntimeException: Exception!
```

12. handle and handleAsync

- **Purpose:** Applies a function to the result or exception of the `CompletableFuture`. The synchronous variant is `handle`, and the asynchronous variant is `handleAsync`.
- **Use Case:** Useful when you need to process the result of a computation regardless of its success or failure, such as optionally transforming the result or providing a default in case of an exception.

```
CompletableFuture<String> handleFuture = CompletableFuture.supplyAsync(() -> {
    if (true) throw new RuntimeException("Handle Exception!");
    return "Handled";
})
.handle((result, ex) -> ex != null ? "Recovered: " + ex.getMessage() : result);

handleFuture.thenAccept(System.out::println); // Prints: Recovered: java.lang.RuntimeException: Handle Exception!
```

```
CompletableFuture<String> handleAsyncFuture = CompletableFuture.supplyAsync(() -> {
    if (true) throw new RuntimeException("Async Handle Exception!");
    return "Async Handled";
})
.handleAsync((result, ex) -> ex != null ? "Async Recovered: " + ex.getMessage() : result);

handleAsyncFuture.thenAccept(System.out::println); // Prints: Async Recovered: java.lang.RuntimeException: Async Handle Exception!
```

Summary

- **Synchronous vs. Asynchronous:** For each operation (`thenApply`, `thenAccept`, `thenRun`, `thenCombine`) there's an asynchronous variant (`thenApplyAsync`, `thenAcceptAsync`, `thenRunAsync`, `thenCombineAsync`) that can execute its task in a separate thread, making it suitable for longer-running operations.
- **Chaining Computations:** These methods enable chaining multiple stages of computation, transforming results, and combining the outcomes of independent computations in a fluent and readable manner.
- **Error Handling:** Methods like `exceptionally` and `handle` provide mechanisms for dealing with errors that may occur during the asynchronous computations, ensuring resilience and robustness in asynchronous logic.

Counter-intuitive Behaviours and How to Address Them

The `CompletableFuture` API in Java is a powerful mechanism for managing asynchronous operations. However, its flexibility can sometimes lead to counterintuitive behaviours, subtle bugs, and performance issues. Understanding these aspects is crucial for developers to effectively use and debug `CompletableFuture`. Let's dive into each point.

Misuse of CompletableFuture Leading to Subtle Bugs and Performance Issues

- **Blocking Calls Inside CompletableFuture:** Using `get()` or `join()` within a `CompletableFuture`'s chain can block the asynchronous execution, negating the benefits of non-blocking code.
Solution: Replace blocking calls with non-blocking constructs like `thenCompose` for chaining futures or `thenAccept` for handling results.
- **Ignoring Returned Futures:** Not handling the `CompletableFuture` returned by methods like `thenApplyAsync` can lead to unobserved exceptions and behavior that does not execute as expected.
Solution: Always chain subsequent operations or attach error handling (e.g. `exceptionally` or `handle`) to every `CompletableFuture`.

Debugging Challenges in Asynchronous Code

- **Stack Traces Lack Context:** Exceptions in asynchronous code can have stack traces that don't easily lead back to the point where the async operation was initiated.

Strategies:

- Use `handle` or `exceptionally` to catch exceptions within the future chain and add logging or breakpoints.
- Consider wrapping asynchronous operations in higher-level methods that catch and log exceptions, providing more context.

Strategies to Identify and Fix Common Issues

- **Consistent Error Handling:** Attach an `exceptionally` or `handle` stage to each `CompletableFuture` to manage exceptions explicitly.
- **Avoid Common Pitfalls:** For example – executing long-running or blocking operations in `supplyAsync` without specifying a custom executor. This can lead to saturation of the `common fork-join pool`.
Solution: Use a custom executor for CPU-bound tasks to prevent interference with the global common fork-join pool.
- **Debugging Asynchronous Chains:** Break down complex chains of `CompletableFuture` operations into smaller parts. Test each part separately to isolate issues.

Tools and Techniques for Debugging CompletableFuture Chains

- **Logging:** Insert logging statements within `completion` stages (e.g., after `thenApply`, `thenAccept`) to trace execution flow and data transformation.
- **Visual Debugging Tools:** Some IDEs and tools offer visual representations of `CompletableFuture` chains, which can help in understanding the flow and identifying where the execution might be hanging or failing.
- **Custom Executors for Monitoring:** Use custom executors wrapped with logging or monitoring to track task execution and thread usage. This is particularly useful for identifying tasks that run longer than expected.
- **Async Profiling:** Tools like `async-profiler` can help identify hotspots and thread activity specific to asynchronous operations.

53. Threads comparison

1. Computations -> Platform threads
2. Super Real Time -> Webflux 100%, VirtualThreads 90-95%, CompletableFuture, Future, etc
3. InputOutput -> VirtualThreads (**project Loom and JDK 21**)

1. Platform → Computation
2. SRT → WebFlux
3. IO → VT

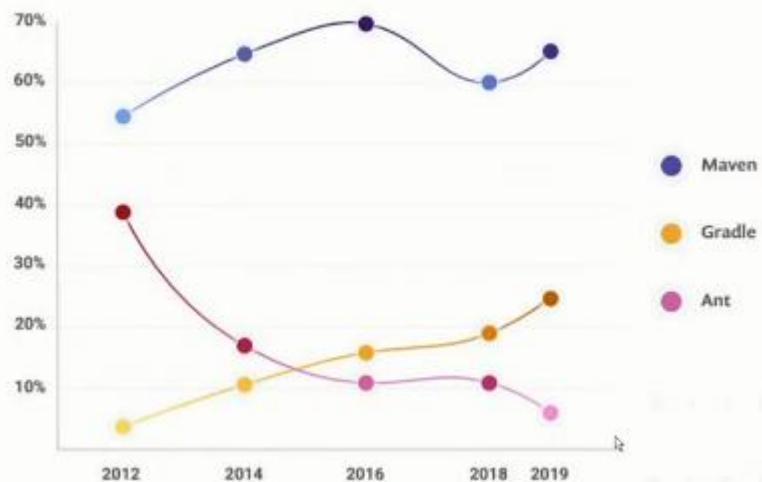
Spring Webflux is substituting Spring MVC. You'd better use only one of them in a microservice!

54. Build systems

Java build systems

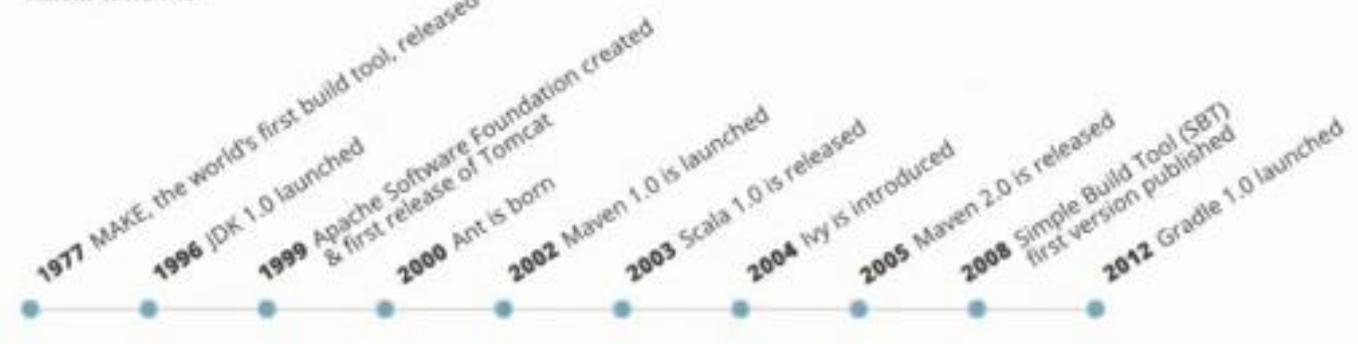
Build systems adoption

Build tool usage since 2012



THE EVOLUTION OF BUILD TOOLS: 1977 - 2013 (AND BEYOND)

Visual timeline



Main features

- The three most popular build systems are Ant, Maven and Gradle
- Build systems automate the process of building distribution artifacts of a project by providing a mechanism for triggering a build process
- A build process can be as simple as:
 - compile the source code with **javac**
 - run the **JUnit** tests
 - Build a **JAR / WAR** file for the project
 - Copy the JAR file and library JARs to a zip file

Jar = uberjar = stand alone = може да се стартира от java-.. = в него има сървър

War = webarchive, използва се при Servlet-и за деплоидване на проекта ни **на външен сървър** като TomCat/TomEE servers.

Features of a build system

- Build systems provide a common set of features such as:
 - Incremental compilation (i.e. compiling only changed sources) - подобрява бързодействието при разработка
 - Management of different profiles (i.e. development and production)
 - Managing project versions
 - Managing project resources (i.e. properties files, images, etc.)
 - Storing and retrieving third-party libraries in a central repository - автоматично - най-популярното е централното Maven repository / хранилище
 - Rich ecosystem of plug-ins
 - A declarative way to describe build steps

Which of the following **is not a capability** of Maven and Gradle build systems ?

- incremental compilation
- plugin management
- project versioning
- code optimization

Ant overview

Info

- Ant was created as a replacement of the MAKE build tool

- Written in Java and best suited for Java projects
- Uses XML files to manage the build process
- The default build file used by Ant is called **build.xml**
- A build file contains an Ant project
- An Ant project contains one or more Ant targets and properties
- Ant targets define the build logic and may contain one or more Ant tasks (like **javac**, **mkdir**, etc.)
- Ant targets may depend on each other thus defining a build order based on a sequence of Ant targets being executed
- Properties are key-value pairs that can be used by targets

Example of build.xml

```
<?xml version="1.0"?>
<project name="Hello" default="COMPILE">
    <target name="COMPILE"
        description="compiles Java source code">
        <mkdir dir="classes"/>
        <javac srcdir"." destdir="classes"/>
    </target>
    ...
</project>
```

The compile target can be executed from the command prompt by navigating to the directory of the build.xml file and running
ant COMPILE

Ant features

Built-in features of Ant include:

- Compilation of Java source code (using **javac**)
- Javadoc generation
- Generation of various archives (such as ZIP, TAR and JAR)
- Managing files and folders
- Sending of email notifications
- Execution of unit tests written in JUnit or TestNG
- Integration with version control systems such as Git or SVN

Ant extensions

Ant tasks can also be provided by third-party extensions to Ant. They are *placed as JAR files* in the **lib** folder of the Ant installation.

One of the most popular extension is **AntContrib** that provides tasks such as:

- **if**: for conditional execution of other tasks in a target
- **for**: for looping within a target
- **switch**: for conditional execution based on a matching value
- **trycatch**: for try-catch logic of execution
- **forget**: run sequence of tasks in a separate thread

Maven

Maven ни предоставя т. наречения Ant plugin, което ни предоставя възможност да изпълняваме неща от Ant.
 Но всъщност не е добре да смесваме build системите.

Overview

Maven aims to solve some of the limitations of the build systems like Apache Ant such as:

- Writing of a complex and verbose XML files in Ant
- Lack of any imposed project structure in Ant
- Lack of built-in dependency management mechanism (although an integration with the Apache Ivy dependency manager is provided in Ant)

Maven:

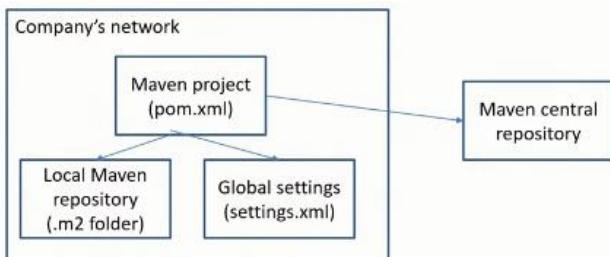
- Maven build files are also based on a XML domain-specific language
- The Maven XML file is less verbose than the one of Ant due to built-in elements for management of dependencies and higher level plug-ins
- The default build file used in Maven is called **pom.xml**
- Можем в един проект да имаме няколко pom.xml файла, които да са обвързани помежду си, а може и пример с отделни микросървиси, които независими отделни pom.xml файл за всеки микросървис.

Features

Main features of Maven include:

- Project dependency management
- Release management and publication
- Multi-module build
- Plug-in system with a number of already existing third-party plugins
- Separation of dependency and plug-in management in the build configuration and Maven repositories
- Possibility to call Ant build targets

Maven architecture

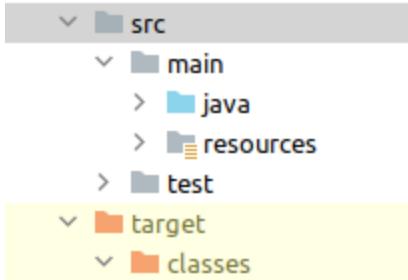


След като майвън зареди глобалните настройки (settings.xml файла който обикновено се намира в инсталационната папка на Maven или *на потребителската директория на операционната система*) и свали от централното репозитори нещата, то се създава Локалното хранилище, което в общия случай се записва в папката **.m2 - на едно от упоменатите 2 места където е и файла с глобалните настройки!** Т.е. локалното Maven repository се използва за кешiranе на тези зависимости, и да не се теглят всеки път, а само при промяна.

Maven project structure

Maven enforced a well-defined project structure:

- **src/main/java**: contains the Java source code of the project
- **src/main/resources**: contains the resource files used by the project
- **src/test/java**: contains the unit/integration tests of the project
- **src/test/resources**: contains the test resource files used by the unit/integration tests
- **target**: default build output directory, **classes** subfolder contains compiled Java source code



Maven project

A Maven build pom.xml file contains a **project** as a start element which may contain a number of elements such as:

- **groupId**, **artifactId** and **version** of the project
- parent Maven project
- Name of the project
- Description of the project
- Packaging type of the project (i.e. JAR or POM(ако не искаме да го билдваме е POM))
- Properties used by the Maven build
- List of dependencies for the project - най-важното предимство - не се налага ръчно да сваляме всеки един JAR на всяка една библиотека
- **build** section with various plug-ins and their configuration used by the build - **даден плъгин може да се изпълни само за определена цел/goal или за всички цели/goals на дадена фаза phase от build lifecycle-a** (clean, compile, verify, install) на проекта
- List of dependency/maven plug-in repositories
- List of child Maven projects (modules)

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd"
xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<modelVersion>4.0.0</modelVersion>
<groupId>io.zerodt</groupId>
<artifactId>profile</artifactId>
<version>1.0.0-SNAPSHOT</version>      SNAPSHOT обозначава, че версията не е стабилен
RELEASE
<properties>
  <assertj.version>3.20.2</assertj.version>
  <compiler-plugin.version>3.8.1</compiler-plugin.version>
  <jacoco.version>0.8.7</jacoco.version>
  <maven.compiler.parameters>true</maven.compiler.parameters>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
  <version.owasp-dependency-check>7.4.3</version.owasp-dependency-check>
</properties>

<repositories>
  <repository>
    <id>central</id>
    <name>Maven Central</name>
    <layout>default</layout>
    <url>https://repo1.maven.org/maven2</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

```
<dependencies>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-hibernate-validator</artifactId>
  </dependency>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-resteasy-jackson</artifactId>
  </dependency>
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-junit5</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct</artifactId>
    <version>${version.mapstruct}</version>
  </dependency>
  <dependency>
    <groupId>org.mapstruct</groupId>
    <artifactId>mapstruct-processor</artifactId>
    <version>${version.mapstruct}</version>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>${assertj.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>${version.apache.commons}</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-collections4</artifactId>
    <version>${version.apache.collection}</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>${quarkus.platform.group-id}</groupId>
      <artifactId>quarkus-maven-plugin</artifactId>
      <version>${quarkus.platform.version}</version>
      <extensions>true</extensions>
      <executions>
        <execution>
          <goals>
            <goal>build</goal>
            <goal>generate-code</goal>
            <goal>generate-code-tests</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```

        </execution>
    </executions>
</plugin>

<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>${compiler-plugin.version}</version>
    <configuration>
        <parameters>${maven.compiler.parameters}</parameters>
        <source>17</source>
        <target>17</target>
        <annotationProcessorPaths>
            <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>${version.mapstruct}</version>
            </path>
        </annotationProcessorPaths>
    </configuration>
</plugin>

<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>${jacoco.version}</version>
    <executions>
        <execution>
            <id>default-instrument</id>
            <goals>
                <goal>instrument</goal>
            </goals>
        </execution>
        <execution>
            <id>default-restore</id>
            <goals>
                <goal>restore-instrumented-classes</goal>
            </goals>
        </execution>
        <execution>
            <id>default-report</id>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
        <execution>
            <id>check</id>
            <phase>verify</phase>
            <goals>
                <goal>check</goal>
            </goals>
            <configuration>
                <rules>
                    <rule implementation="org.jacoco.maven.RuleConfiguration">
                        <element>CLASS</element>
                        <limits>
                            <limit>
                                <counter>LINE</counter>
                                <value>COVEREDRATIO</value>
                                <minimum>100%</minimum>
                            </limit>
                        </limits>
                    </rule>
                </rules>
            </configuration>
        </execution>
    </executions>
</plugin>

```

```
<limit>
    <counter>BRANCH</counter>
    <value>COVEREDRATIO</value>
    <minimum>100%</minimum>
</limit>
</limits>
</rule>
</rules>
</configuration>
</execution>
</executions>
<configuration>
<excludes>
    <exclude>**/*MapperImpl.class</exclude>
    <exclude>**/cms/dto/**/*</exclude>
    <exclude>**/cache/*</exclude>
</excludes>
</configuration>
</plugin>
</plugins>
</build>

<profiles>
<profile>
    <id>native</id>
    <activation>
        <property>
            <name>native</name>
        </property>
    </activation>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-failsafe-plugin</artifactId>
                <version>${surefire-plugin.version}</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>integration-test</goal>
                            <goal>verify</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    <properties>
        <quarkus.package.type>native</quarkus.package.type>
    </properties>
</profile>

<profile>
    <id>owasp</id>
    <activation>
        <property>
            <name>owasp</name>
        </property>
    </activation>
    <configuration>
        <excludes>
            <exclude>**/*MapperImpl.class</exclude>
            <exclude>**/cms/dto/**/*</exclude>
            <exclude>**/cache/*</exclude>
        </excludes>
        <limits>
            <limit>
                <counter>BRANCH</counter>
                <value>COVEREDRATIO</value>
                <minimum>100%</minimum>
            </limit>
        </limits>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</profile>
```

```

</activation>

<build>
  <plugins>
    <plugin>
      <groupId>org.owasp</groupId>
      <artifactId>dependency-check-maven</artifactId>
      <version>${version.owasp-dependency-check}</version>
      <configuration>
        <failBuildOnCVSS>9</failBuildOnCVSS>
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

</profile>

</profiles>

</project>

```

Maven build lifecycle

What does a Maven lifecycle contain? - stages (**phases**) and **goals** (tasks) **executed in each phase**

- Maven works by means of so called **build lifecycle** that defines a sequences of **phases** to be executed.
- **Phases** are the Maven equivalent of Ant target whereby each phase may trigger one or more **goals** that perform the actual work - **даден plugin можем да му кажем да се изпълни в даден phase като му даваме име на задачата(дадена цел):**

Например в compile фазата даден плъгин, който първо ни създава директория задача цел 1, и след това задача цел 2 да е компилация на всички папки от нашето приложение

- A **goal** is a unit of work performed by a Maven plug-in that implements the goal

Maven default lifecycle

Maven provides a default lifecycle that has the following **phases**:

- **validate**
- **generate-sources**
- **process-sources**
- **generate-resources**
- **process-resources**
- **compile**
- **process-test-sources**
- **process-test-resources**
- **test-compile**
- **test**
- **package**
- **verify**
- **install**
- **deploy**

Which of the following Maven phases creates a dependency in the local Maven repository?

- a. compile
- b. package
- c. **install** - install phase installs the built artifact into the local repository for use as a dependency in other projects.
- d. Deploy

`mvn package` This command builds the maven project and packages them into a JAR, WAR, etc.
`./mvnw clean package` Ако използваме wrapper-a.

Maven phases

- A maven phase can be executed with the **mvn** tool
- All phases up to the specific one in the default lifecycle are executed in order - i.e. **each phase include all previous phases in the order above**

<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

Всяка команда включва в себе си предходните команди - считано отгоре надолу

Build Lifecycle

- `validate` - validate the project is correct and all necessary information is available
- `compile` - compile the source code of the project
- `test` - test the compiled source code using a suitable **unit testing** framework. These tests should not require the code be packaged or deployed
- `package` - take the compiled code and package it in its distributable format, such as a JAR.
- `verify` - run any checks on results of **integration tests** to ensure quality criteria are met
- `install` - install the package into the local .m2 repository, **for use as a dependency in other projects locally**
- `deploy` - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

Maven dependencies

- Maven dependencies are other Maven projects along with associated build artifacts (such as JAR files)
- Maven dependencies are managed by Maven repositories
- Maven repositories can be public (like Maven Central which is the default one) or private within an organization
- Други хранилища имплементиращи Maven:
 - Artifactory - Universal Artifact Repository Manager - JFrog;
 - Sonatype Nexus Repository - Binary & Artifact Management

Dependency scopes

Maven dependencies may have a scope associated such as:

- **compile** (*default scope*): dependency is required during compilation
- **provided**: available at compile time but at runtime should be provided by a container - има я библиотеката готова в друга среда и не е необходимо да се свали локално и да бъде част от проекта
- **runtime**: dependency is not required for compilation but it is required at runtime
- **test**: dependency is available for test compilation and execution phases only - библиотеки и зависимости необходими само за изпълнението на нашите юнит тестове (JUNIT, Mockito)
- **system**: JAR file is provided explicitly (i.e. via location on the file system) - например JDBC библиотеката за свързване с база данни можем освен да я декларираме в pom.xml файла, то и да и зададем scope **system**

Maven multi module projects

- A multi module Maven project provides the possibility to build more than one child projects at once in order - всеки от подпроектите има отделен pom.xml файл, и текущия ги групира

Пример за взаимна свързаност

Parent проект

```
<groupId>bg.jug.academy</groupId>
<artifactId>parent</artifactId>
<modules>
<module>../utilities</module>
<module>../services</module>
<module>../persistence</module>
</modules>
```

Child проект utilities

```
<groupId>bg.jug.academy</groupId>
<artifactId>utilities</artifactId>
<parent>
    <groupId>bg.jug.academy</groupId>
    <artifactId>parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <relativePath>../parent</relativePath>  ако го имаме в локалното .m2 хранилище
</parent>
```

How can you define multi module builds in Maven?

- By listing child modules in a <modules> block in the pom.xml file
- By referencing child modules as dependencies in the <dependencies> block in the pom.xml file
- Using a proper Maven module management plugin
- It is not supported

Dependency Management

При свързани проекти използващи едни и същи зависимости - ако декларираме библиотеки в parent главния проект в секция dependencyManagement със scope и версия, то в child проектите няма да е необходимо да декларираме scope и версия, а то само ще си ги взема!!!

```
<dependencyManagement>
<dependencies>
    <dependency>
        <groupId>${quarkus.platform.group-id}</groupId>
        <artifactId>${quarkus.platform.artifact-id}</artifactId>
        <version>${quarkus.platform.version}</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
</dependencies>
</dependencyManagement>
```

Разбира се, по-добър вариант е да изнесем версиите в <properties> секцията:

```
<properties>
...
</properties>
```

Exclusions

Когато искаме да изключим дадена подбиблиотека да се свали и използва локално, то ѝ забраняваме с `<exclusion>`

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-junit5</artifactId>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.junit.platform</groupId>
      <artifactId>junit-platform-launcher</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Maven profiles

Maven profiles provide the possibility to create different build logic based on a specified profile (i.e. development or production)

mvn -P production compile

<build> за даден профил в pom.xml файла - работи само за дадения профил, в т.ч. и плъгините:

```
<profiles>
  <profile>
    <id>production</id>
    <activation>
      <property>
        <name>native</name>
      </property>
    </activation>

    <build>
      <plugins>
        <plugin>
          <artifactId>maven-failsafe-plugin</artifactId>
          <version>${surefire-plugin.version}</version>
          <executions>
            <execution>
              <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Maven archetypes

- Maven archetypes provide predefined “template” projects than can be generated - like Spring project, Hibernate project, etc.
- Maven archetypes can be listed (and generated) with the **archetype: generate** goal

mvn archetype:generate

Maven plug-ins

- Maven provides a rich ecosystem of plug-ins that can be used in Maven builds
- A plug-in provides one or more MOJOS (Maven old Java objects) that implement different plug-in goals
 - как сами можем да имплементираме даден плъгин
- A plug-in goal can be triggered directly from the command line independently from a build using the following syntax:

mvn [plugin-name]:[goal-name]

For example:

mvn compiler:compile

Core plug-ins include:

- **compiler**: for compilation of Java source code
- **surefire**: for unit test execution
- **jar**: for building a JAR file from the current project
- **javadoc**: for Javadoc generation
- **dependency**: for dependency manipulation

Full list: <https://maven.apache.org/plugins/index.html>

<build> в главното място в pom.xml файла - работи за всички профили, в т.ч. и плъгините:

<project>

```
<build>
  <plugins>
    <plugin>
      <groupId>${quarkus.platform.group-id}</groupId>
      <artifactId>quarkus-maven-plugin</artifactId>
      <version>${quarkus.platform.version}</version>
      <extensions>true</extensions>
      <executions>
        <execution>
          <goals>
            <goal>build</goal>
            <goal>generate-code</goal>
            <goal>generate-code-tests</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>${compiler-plugin.version}</version>
      <configuration>
        <parameters>${maven.compiler.parameters}</parameters>
        <source>17</source>
        <target>17</target>
        <annotationProcessorPaths>
          <path>
            <groupId>org.mapstruct</groupId>
            <artifactId>mapstruct-processor</artifactId>
            <version>${version.mapstruct}</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>${surefire-plugin.version}</version>
  <configuration>
    </configuration>
</plugin>

</plugins>
</build>

</project>
```

Taking a full list of dependencies on maven

Отиваме на директорията на Java приложението, което разработваме, и пускаме следната терминална команда:

Изпълни ми командата и запиши резултата в tree.txt файл

mvn dependency:tree > tree.txt

```
NFO] Scanning for projects...
NFO]
NFO] -----< bg.jug.academy:application >-----
NFO] Building Example application 0.0.1-SNAPSHOT
NFO] -----[ jar ]-----
NFO]
NFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ application ---
NFO] bg.jug.academy:application:jar:0.0.1-SNAPSHOT
NFO] +- org.junit.jupiter:junit-jupiter:jar:5.9.2:test
NFO] |  +- org.junit.jupiter:junit-jupiter-api:jar:5.9.2:test
NFO] |  |  +- org.opentest4j:opentest4j:jar:1.2.0:test
NFO] |  |  \- org.junit.platform:junit-platform-commons:jar:1.9.2:test
NFO] |  +- org.junit.jupiter:junit-jupiter-params:jar:5.9.2:test
NFO] |  \- org.junit.jupiter:junit-jupiter-engine:jar:5.9.2:test
NFO] +- org.junit.platform:junit-platform-launcher:jar:1.9.3:test
NFO] |  +- org.junit.platform:junit-platform-engine:jar:1.9.3:test
NFO] |  \- org.apiguardian:apiguardian-api:jar:1.1.2:test
NFO] +- org.mockito:mockito-core:jar:5.2.0:test
NFO] |  +- net.bytebuddy:byte-buddy:jar:1.14.1:test
NFO] |  +- net.bytebuddy:byte-buddy-agent:jar:1.14.1:test
NFO] |  \- org.objenesis:objenesis:jar:3.3:test
NFO] +- org.mockito:mockito-junit-jupiter:jar:5.2.0:test
NFO] \- org.seleniumhq.selenium:selenium-java:jar:4.10.0:test
NFO]   +- org.seleniumhq.selenium:selenium-api:jar:4.10.0:test
NFO]   +- org.seleniumhq.selenium:selenium-chrome-driver:jar:4.10.0:test
NFO]   |  +- com.google.auto.service:auto-service-annotations:jar:1.0.1:test
NFO]   |  +- com.google.auto.service:auto-service:jar:1.0.1:test
NFO]   |  |  \- com.google.auto:auto-common:jar:1.2:test
NFO]   |  +- com.google.guava:guava:jar:31.1-jre:test
NFO]   |  +- com.google.guava:failureaccess:jar:1.0.1:test
NFO]   |  +- com.google.guava:listenablefuture:jar:9999.0-empty-to-avoid-conflict-with
NFO]   |  +- com.google.code.findbugs:jsr305:jar:3.0.2:test
NFO]   |  +- org.checkerframework:checker-qual:jar:3.12.0:test
NFO]   |  +- com.google.errorprone:error_prone_annotations:jar:2.11.0:test
NFO]   |  \- com.google.j2objc:j2objc-annotations:jar:1.3:test
NFO]   +- org.seleniumhq.selenium:selenium-chromium-driver:jar:4.10.0:test
NFO]   +- org.seleniumhq.selenium:selenium-json:jar:4.10.0:test
NFO]   \- org.seleniumhq.selenium:selenium-manager:jar:4.10.0:test
NFO] +- org.seleniumhq.selenium:selenium-devtools-v112:jar:4.10.0:test
NFO] +- org.seleniumhq.selenium:selenium-devtools-v113:jar:4.10.0:test
```

In Eclipse IDE installed libraries/dependencies of the project are shown with the full path to the .m2 local repository

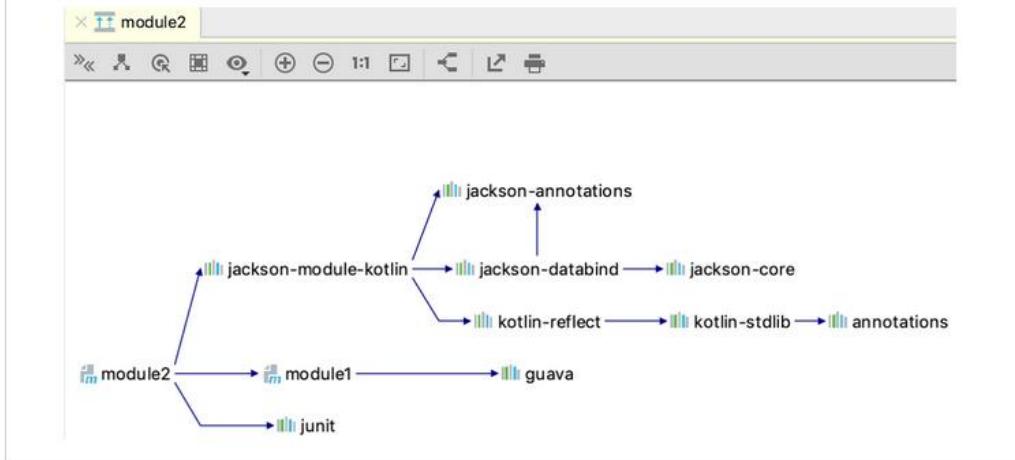


View Maven dependencies as a diagram - ultimate IntelliJ only

In any pom.xml file , right click then

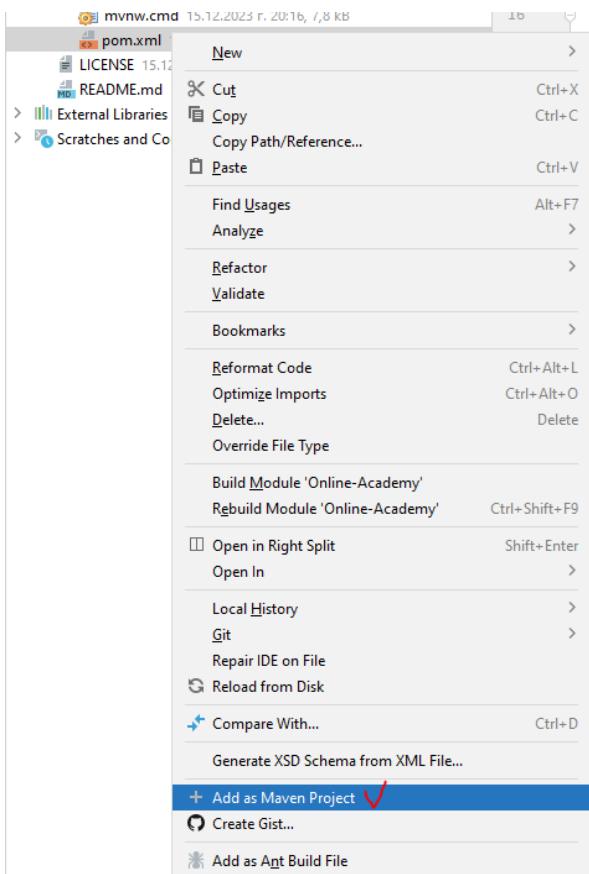


2. In the diagram window, IntelliJ IDEA displays the sub project and all its dependencies including the transitive ones.

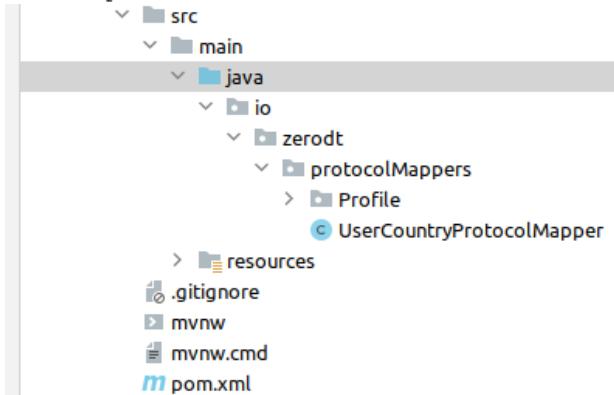


Load a microservice project in IntelliJ Ultimate

Ако имаме незареден проект, отиваме на **pom.xml** файла, десен бутон



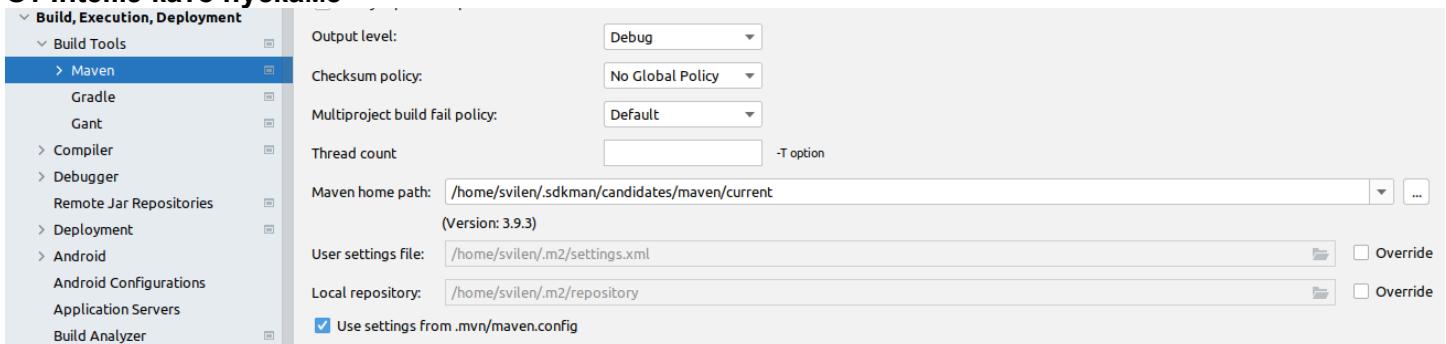
и го зарежда с боядисани в синьо (и зелено папки) јава както си е нормално.

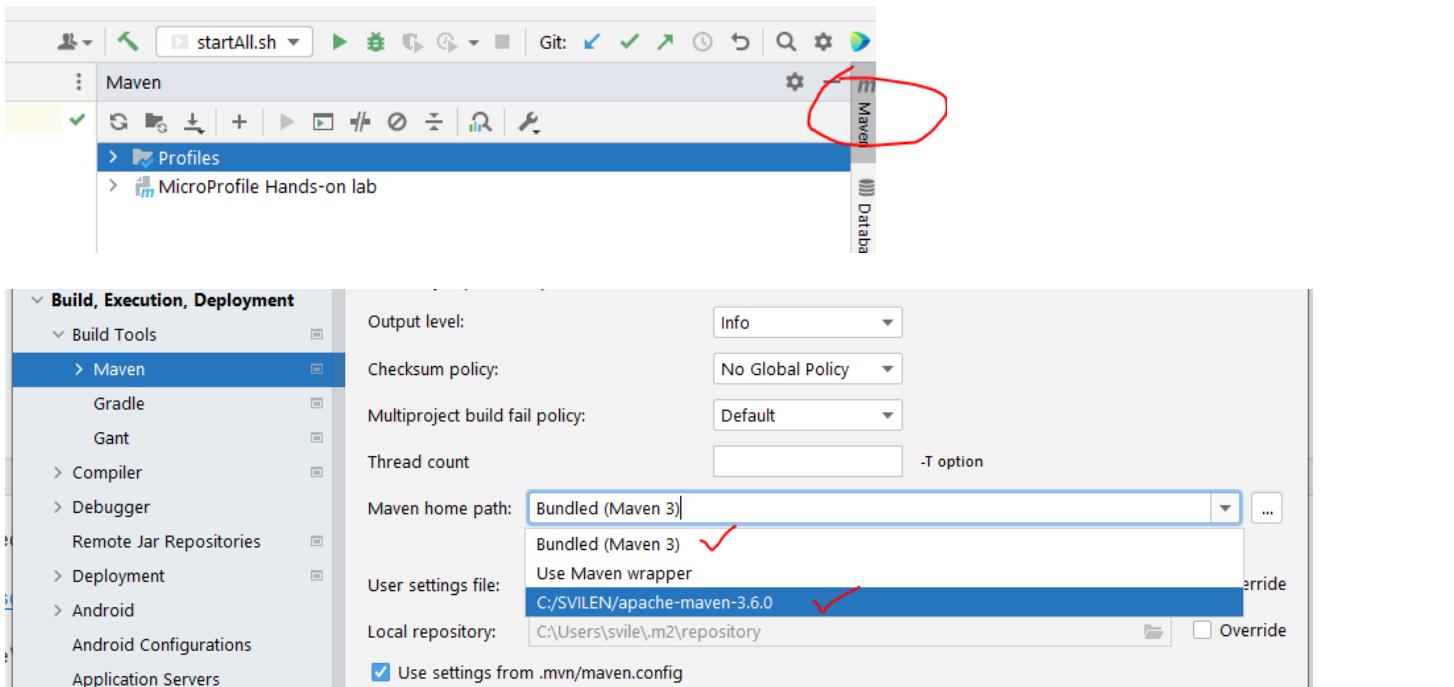


Wrapper

Без wrapper-a:

От IntelliJ като пускате





От терминала като пускаме

`mvn -version` показва версията на maven

`mvn` команда само изпълнява/сваля библиотеките

Трябва в текущата директория на терминала да има pom.xml, за да тръгнат нещата

`mvn quarkus:dev` we start the app with live re-load

`mvn compile quarkus:dev` we start the app with live re-load

mvn -version - за момента ми дава също и текущата версия на java jdk (в IntelliJ се опреснява след рестарт на IntelliJ)

`java -version`

`javac -version`

`mvn dependency:tree`

`docker --version`

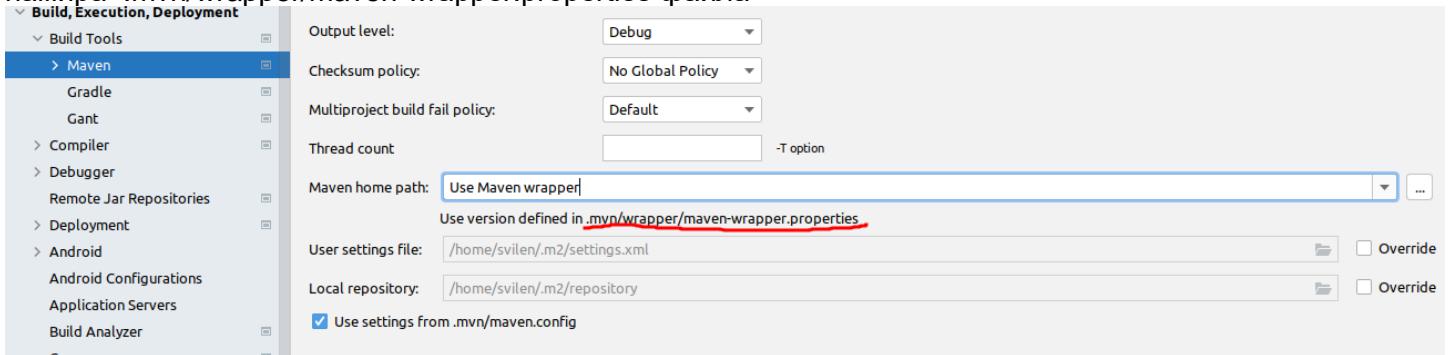
`mvn test` в проекта/директорията - от терминала

С wrapper-a:

От IntelliJ като пускаме –

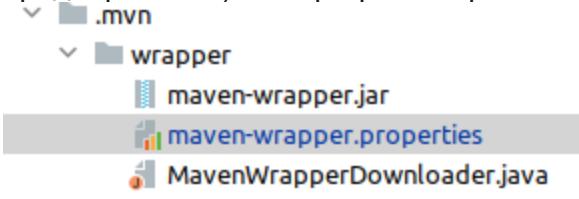
В случая с много микросървици обаче - този вариант няма да е ок защото не знаем къде се

намира .mvn/wrapper/maven-wrapper.properties файла



От терминала:

maven wrapper-ът е tool-че, което ти сваля maven локално (така че да не ти се налага да го правиш предварително). и от properties файла взима коя версия да свали



maven-wrapper.properties

```
distributionUrl=https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/3.9.3/apache-maven-3.9.3-bin.zip  
wrapperUrl=https://repo.maven.apache.org/maven2/io/takari/maven-wrapper/0.5.6/maven-wrapper-0.5.6.jar
```

Помът казва каква минимална версия трябва да имаш инсталриана. Без да се интересува (помът) как си я инсталирал - дали ръчно, дали с SDKMan или е дошла от mvnw.

```
<properties>  
  <compiler-plugin.version>3.8.1</compiler-plugin.version>  
</properties>  
  
<build>  
  <plugins>  
    <plugin>  
      <groupId>${quarkus.platform.group-id}</groupId>  
      <artifactId>quarkus-maven-plugin</artifactId>  
      <version>${quarkus.platform.version}</version>  
      <extensions>true</extensions>  
      <executions>  
        <execution>  
          <goals>  
            <goal>build</goal>  
            <goal>generate-code</goal>  
            <goal>generate-code-tests</goal>  
          </goals>  
        </execution>  
      </executions>  
    </plugin>  
    <plugin>  
      <artifactId>maven-compiler-plugin</artifactId>  
      <version>${compiler-plugin.version}</version>  
      <configuration>  
        <parameters>${maven.compiler.parameters}</parameters>  
        <source>17</source>  
        <target>17</target>  
        <annotationProcessorPaths>  
          <path>  
            <groupId>org.mapstruct</groupId>  
            <artifactId>mapstruct-processor</artifactId>  
            <version>${version.mapstruct}</version>  
          </path>  
        </annotationProcessorPaths>  
      </configuration>  
    </plugin>  
  </plugins>  
</build>
```

Install the wrapper

```
mvn wrapper:wrapper
```

<https://maven.apache.org/wrapper/>

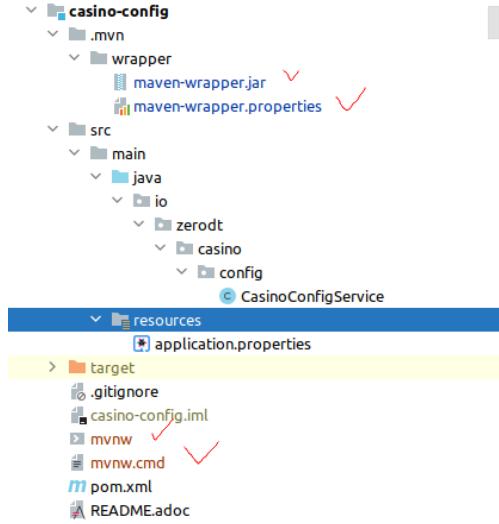
Тези 2 реда ги има в записките

Install the wrapper

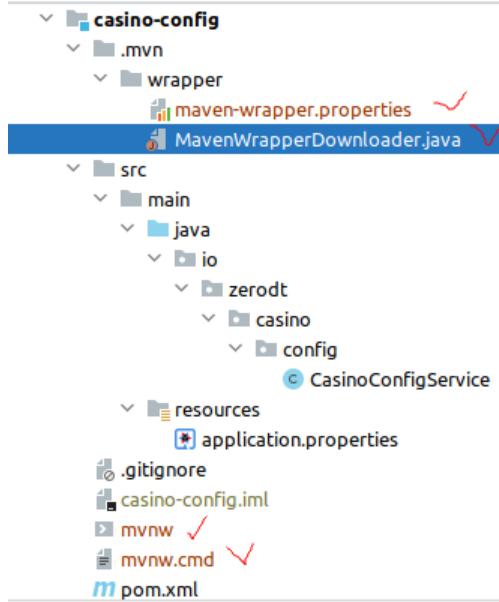
```
mvn wrapper:wrapper
```

By default, the Maven Wrapper JAR archive is added to the using project as small binary file `.mvn/wrapper/maven-wrapper.jar`. It is used to bootstrap the download and invocation of Maven from the wrapper shell scripts.

Командата инсталира следните файлове



If your project is not allowed to contain binary files like this, you can use the source distribution of the maven wrapper which adds a file `.mvn/wrapper/MavenWrapperDownloader.java` file instead:



Run the goals

```
./mvnw clean install
```

The Maven Wrapper is an excellent choice for projects that need a specific version of Maven (or for users that don't want to install Maven at all). Instead of installing many versions of it in the operating system, we can just use the project-specific wrapper script.

./mvnw quarkus:dev - това е терминалната команда с wrapper-a

Should *mvnw* files be added to our projects?

The short answer is no. ***mvnw* files are not necessarily part of our projects. However, including them could be beneficial.** For example, it will allow anyone cloning our project to build it without installing Maven.

Gradle

Overview

- В Gradle много по-лесно можем да създаваме задачи в сравнение с Maven - както в Ant лесно можем да създаваме задачи
- Gradle is a build tool based on the concepts of Ant and Maven
- Gradle build files are based on a Groovy domain-specific language
- The default Gradle build file is called **build.gradle**
- Gradle enforces the same directory conventions as Maven
- Dependencies are defined using the same format
- Gradle builds may also step on existing Maven or Ant (Ivy) repositories
- The **java** Gradle plug-in emulates the Maven default lifecycle
- A Gradle project consists of one or more Gradle tasks
- Gradle provides a number of predefined tasks
- Properties may be specified in a **gradle.properties** file
- Subprojects (**modules**) can be specified in a **settings.gradle** file

Example:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    baseName = 'example'
    Version = `1.0.0-SNAPSHOT`
}

dependencies {
    compile 'junit:junit:4.12'
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.hateoas:spring-hateoas'
    runtimeOnly 'com.h2database:h2'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    implementation 'org.mapstruct:mapstruct:1.5.1.Final'
    compileOnly 'org.mapstruct:mapstruct-processor:1.5.1.Final'
```

```
annotationProcessor 'org.mapstruct:mapstruct-processor:1.5.1.Final'  
}  
  
tasks.named('test') {  
    useJUnitPlatform()  
}
```

implementation = compile + runtime или това което трябва да се изпълни на production средата

Example Gradle task with Ant

An Ant task can be called from a Gradle task as follows - например при миграция на Ант таскове към Градле таскове, можем така да wrap-нем Ант тасковете:

```
task zip {  
    doLast {  
        ant.zip(destfile: 'archive.zip') {  
            fileset(dir: 'src') {  
                include(name: '**.xml')  
                include(name: '**.java')  
            }  
        }  
    }  
}
```

Започване на нов проект през терминала

Можем да инициализираме проекта със следната терминална команда:

gradle init

```
D:\projects\java_academy\workspace\testproject>gradle init  
Starting a Gradle Daemon (subsequent builds will be faster)
```

```
Select type of project to generate:  
1: basic  
2: application  
3: library  
4: Gradle plugin  
Enter selection (default: basic) [1..4] 2
```

```
Select implementation language:  
1: C++  
2: Groovy  
3: Java  
4: Kotlin  
5: Scala  
6: Swift  
Enter selection (default: Java) [1..6] 3
```

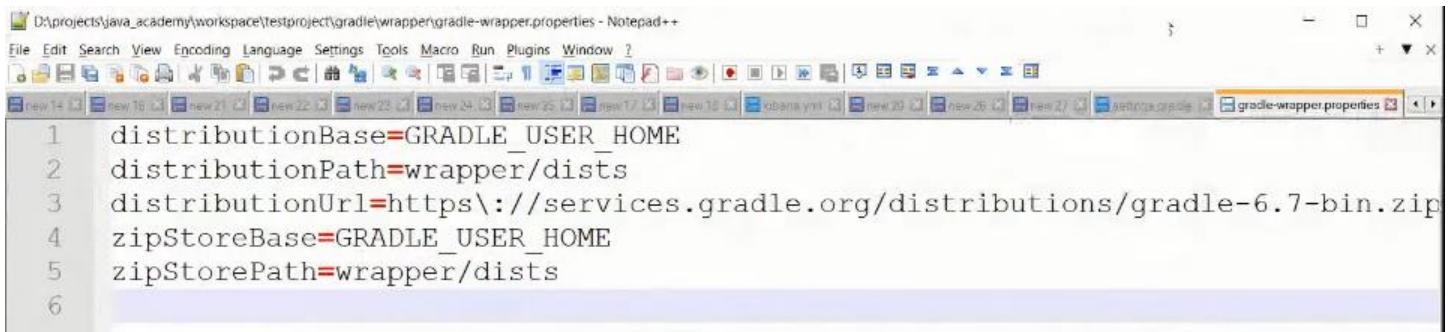
И разбира се - винаги задаваме и home директорията на Gradle/Maven

Можем да пуснем и терминална команда:

gradle --version

gradle run

gradlew (или т.нар gradle wrapper) е просто един допълнителен скрипт за съответната операционна система (.bat, ...) - лесно да се менажират версии на gradle автоматично или по подразбиране(версия 6.7 в примера по-долу). **Затова gradlew е предпочитаема опция.**



```
1 distributionBase=GRADLE_USER_HOME
2 distributionPath=wrapper/dists
3 distributionUrl=https://services.gradle.org/distributions/gradle-6.7-bin.zip
4 zipStoreBase=GRADLE_USER_HOME
5 zipStorePath=wrapper/dists
6
```

gradle clean

Exclude some sublibrary

```
implementation('org.apache.pdfbox:pdfbox:2.0.28') {
    exclude group: 'commons-collections', module: 'commons-collections'
}
```

Working with modules

Parent project GradleExample - settings.gradle file

```
9
10rootProject.name = 'GradleExample'
11include('lib')
12
```

Child project lib

apply plugin: 'java'

```
repositories {
    mavenCentral()
}
```

```
jar {
    baseName = 'example'
    Version = `1.0.0-SNAPSHOT`
}
```

```
dependencies {
    compile 'junit:junit:4.12'
}
```

```
task hello {
    doLast {
        println 'BGJUG'
    }
}
```

```
hello.doLast {
    println 'BGJUG academy'
}
```

```
hello.doFirst {
```

```
    println 'muahaha'  
}
```

How can you define multi module builds in Gradle ?

- a. **Using settings.gradle file to define submodules**
- b. Using build.gradle file to define submodules
- c. By referencing child modules as dependencies in the <dependencies> block
- d. Using a proper Gradle module management plugin

Рънни подпроекта **lib**, задачата му **hello**

gradlew :lib:hello

Билдни конкретния подпроект

gradlew :lib:build

Билдни парент проекта и всички подпроекти

gradlew build

Показва всички таскове/properties c gradle wrapper (gradlew)

gradlew tasks

gradlew properties

Working with profiles

gradlew -Pdevelopment build

Ако има пропърти development

```
50hello.doLast {  
51     if(project.hasProperty('development')) {  
52         println 'BGJUG academy'  
53     }  
54}
```

Gradle tasks and phases

Which gradle task can be used to **assemble** and **test** a project ?

- a. **build**
- b. classes
- c. test
- d. run

The **build task** uses the source code and its dependencies to build the app. As seen in the output, the **build task do: compiles, assembles, tests, and checks the code.**

С определени настройки, командата `./gradlew clean build` генерира и .jar файл. (Виж Spring + Web Notes -> Demo AutoConfiguration....)

When you run the **build** command you'll see output like this.

This means that your application was assembled and any tests passed successfully. Gradle will have created a build directory if it didn't already exist, containing some useful outputs you should be aware of.

1. the **classes directory** contains compiled .class files, as a result of running the Java compiler against your application Java source files
2. the **libs directory** contains a generated jar file, an archive with all your compiled classes inside **ready to be executed or published**

3. the **reports** directory contains an HTML report summarising your test results. If your build fails, then consult this report to see what went wrong.

https://docs.gradle.org/current/userguide/build_lifecycle.html

A Gradle build has three distinct phases. Gradle runs these phases in order: first initialization, then configuration, and finally execution.

Initialization

- Detects the [settings file](#).
- Evaluates the settings file to determine which projects and included builds participate in the build.
- Creates a [Project](#) instance for every project.

Configuration

- Evaluates the build scripts of every project participating in the build.
- Creates a task graph for requested tasks.

Execution

- Schedules and executes each of the selected tasks in the order of their dependencies.

[Example](#) Groovy style

```
settings.gradle
rootProject.name = 'basic'
println 'This is executed during the initialization phase.'


build.gradle
println 'This is executed during the configuration phase.'

tasks.register('configured') {
    println 'This is also executed during the configuration phase, because :configured is
used in the build.'
}

tasks.register('test') {
    doLast {
        println 'This is executed during the execution phase.'
    }
}

tasks.register('testBoth') {
    doFirst {
        println 'This is executed first during the execution phase.'
    }
    doLast {
        println 'This is executed last during the execution phase.'
    }
    println 'This is executed during the configuration phase as well, because
:testBoth is used in the build.'
}
```

[Spring dependency management for Gradle](#)

build.gradle

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.2.0'
    id 'io.spring.dependency-management' version '1.1.4' //грижи се за съвместимост на
версии
```

```

}

group = 'com.example'
version = '0.0.1-SNAPSHOT'

java {
    sourceCompatibility = '17'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf' //няма версия
    и това е яко
    implementation 'org.springframework.boot:spring-boot-starter-web' //няма версия и това
    е яко

    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
    useJUnitPlatform()
}

```

55. JVM innerworkings

55.1. Hotspot JVM

Virtual machines

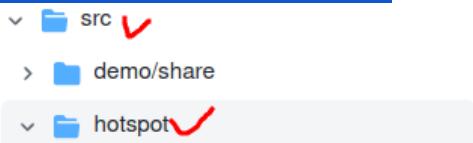
A typical virtual machine for an interpreted language provides:

- Compilation of source language **into** VM specific **bytecode** - **компилира се до bytecode** (съвкупност от компилирани Java класове)
- Data structures to contains instructions and operations (the data the instructions process) - структури от данни за това какви са операциите и процесорните инструкции, които се поддържат от **bytecode-а** на JavaVirtualMachine (JVM)
- A **call stack** for function call operations
- An '**Instructor Pointer**' (IP) pointing to the next processor instruction to execute
- A **virtual ‘CPU’** - the instruction dispatcher that:
 - *Fetches* the next instruction (addressed by the instruction pointer)
 - *Decodes* the operands
 - *Executes* the instruction

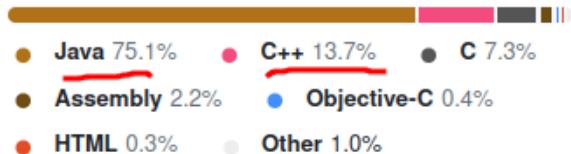
The HotSpot JVM

Кода на JVM го има в интернет:

<https://github.com/openjdk/jdk/>



Languages



HotSpot is the standard JVM distribution of Oracle Corp. that provides:

- **bytecode execution**
 - using an **interpreter - bytecode here**,
 - *На практика в някои случаи минаваме и през тази стълка!!* - two **runtime compilers (JIT** (Just-In-Time) compilers) for optimization - например ако един метод се вика 50 000 пъти, то JVM преценява дали да го **компилира до машинен код**. Като компилирания машинен код е по-бърз от компилирания bytecode!! Client and server - C1 and C2 in JVM. Now they work together and not separately
 - **Client JIT** - важно приложението да стартира бързо, а цялото зареждане допълнение
 - **Server JIT** - за големи сървърни приложения. JVM по време на компилиране прави много оптимизации и стартира по-бавно. Така че когато е заредило вече приложението, то логиката в него пък да се изпълнява по-бързо
 - and **On-Stack Replacement** - JVM може да прецени да не изпълнява компилирания машинен код, и вместо това да изпълни компилирания bytecode. Т.нар. деоптимизация.

• storage allocation and garbage collection

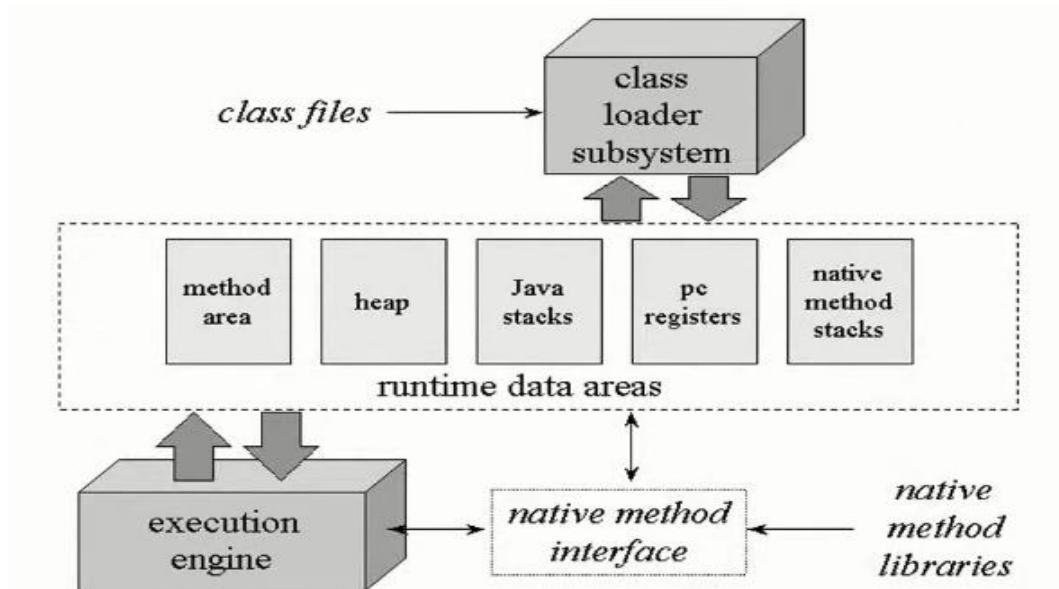
Алокиране на памет за обектите, които създаваме в нашето приложение. Т.нар. HEAP памет се заделя при стартиране на нашето Java приложение.

Колко памет да използва, колко минимална и максимална памет да използва.

Бързодействието на самата JVM машина се определя от garbage collection-а - колко бързо можем да освободим памет без да нарушим изпълнението на нашата програма.

- **runtimes** - start up, shut down, class loading, threads, interaction with OS and others

Architecture of the HotSpot JVM



1. Class-loading

Зареждане до класфайловете от .java файлове

Three phases of class-loading:

- **Loading**

С коя версия на Java compiler е компилиран дадения .java файл. Това се записва в minor и major versions.

Constant_pool - Информация/масив за всички константи, които имаме в дадения клас

Access_flags - public, private, protected

Interfaces - масив с всички интерфейси на класа

Fields - масив с всички полетата на класа

Methods - масив с всички методи на класа

Attributes - масив с допълнителни атрибути, които може да сме дефинирали в класа

```
ClassFile {
    u4          magic;
    u2          minor_version;
    u2          major_version;
    u2          constant_pool_count;
    cp_info     constant_pool[constant_pool_count-1];
    u2          access_flags;
    u2          this_class;
    u2          super_class;
    u2          interfaces_count;
    u2          interfaces[interfaces_count];
    u2          fields_count;
    field_info   fields[fields_count];
    u2          methods_count;
    method_info  methods[methods_count];
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class Data

Run-Time Constant Pool

string constants
numeric constants
class references
field references
method references
name and type
Invoke dynamic



- **Linking** - създават се връзки между съответните класфайлове в JVM
- **Initialization** - например инициализация на статични блокове, на константи

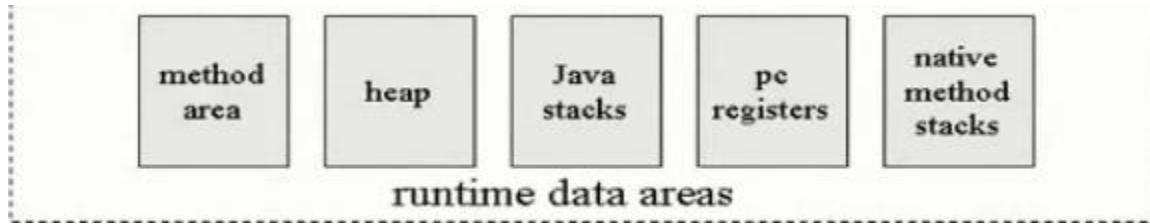
```
static int volume(int width,
                  int depth,
                  int height) {
    int area = width * depth;
    int volume = area * height;
    return volume;
}
```



```
0 iload_0
1 iload_1
2 imul
3 istore_3
4 iload_3
5 iload_2
6 imul
7 istore_4
9 iload_4
11 ireturn
```

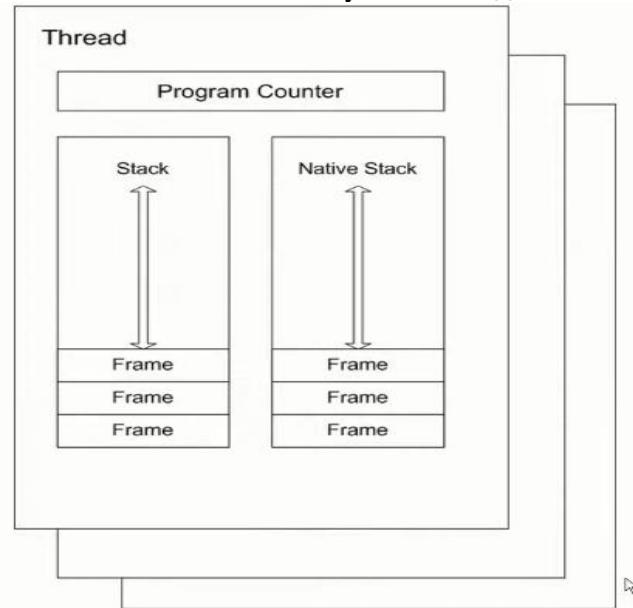
2.Runtime data areas

Main info



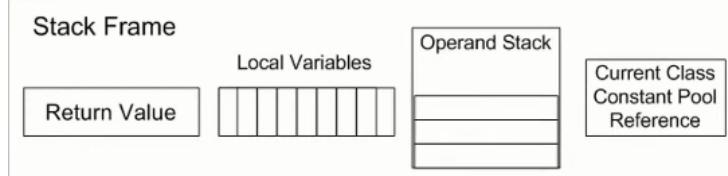
В JVM се зареждат различни области, които се използват от приложението ни. Като например:

- Област за HEAP паметта - за аллокиране на обектите от приложението
- Method area - описват се методите, които се изпълняват
- Java stacks - създават се стекови от нишките, които се стартират в рамките на приложението
- **Program counter registers**
- Stacks за изпълнение на native методи - JVM отдолу в голяма степен е имплементирана на C++, така че в някои случаи може да се викат и методи на C++



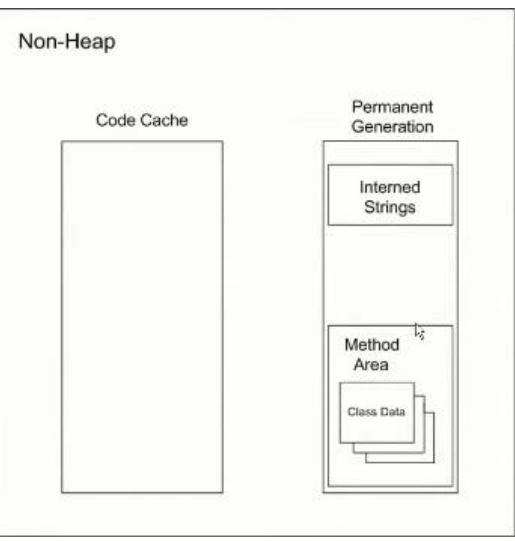
- Отделно JVM може да реши да използва и **native threads (на ниво ОС)** и за тях се създава **отделен Stack!**

За даден стек фрейм имаме:



Before JDK 8

Permanent generation - отделна област за съхраняване на информация за string-овите константи и за методите заредени от JVM



As of JDK 8

Permanent generation is part of the HEAP

HEAP

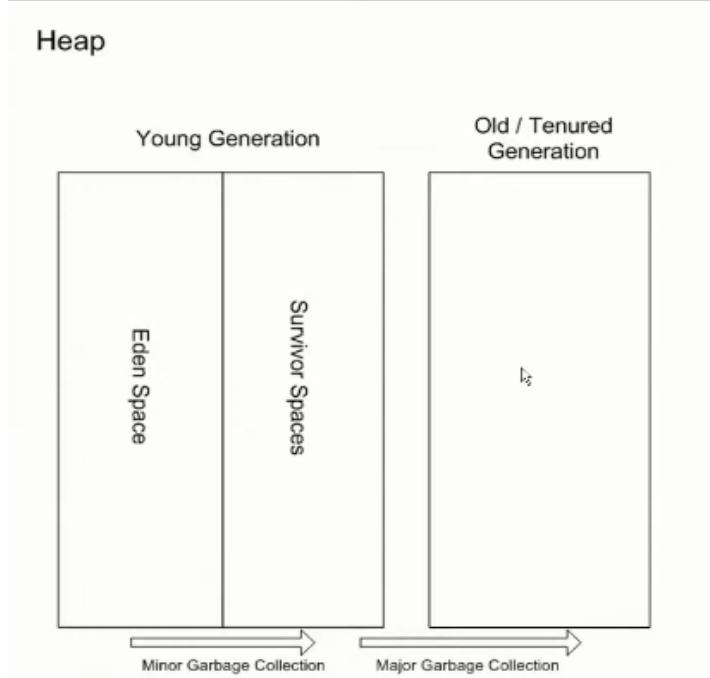
Young generation - пазят се тук обекти, които живеят кратко

Old / Tenured Generation - например за логер, който живее през цялото време.

Обекти, за които JVM прецени че живеят по-дълго от очакваното се местят от Young generation към Old / Tenured Generation!

Обектите от Young generation се изчистват по-лесно и бързо от garbage collector-а, без да се прави задълбочен анализ от garbage collector-а.

Чистенето на обекти от Old / Tenured Generation става чрез паузиране, което за някои приложения е ключово. Затова трябва да се стремим чистенето на обекти от Old / Tenured Generation да бъде минимално по дължина време!!!



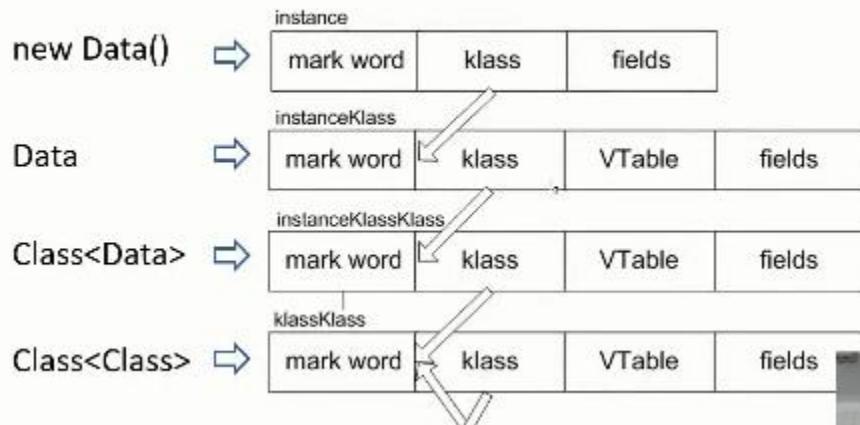
При създаване на нов обект:

VTable = VirtualTable - при полиморфизъм влиза в употреба

Създават се няколко масива в хийпа:

- един за инстанцирания обект
- един за типа на обекта - от кой клас е - from reflection .getClass
- Class Генерик от тип нашия клас
- Class Генерик от тип Class

- Heap memory:



И така за всеки един обект!!!

`mark word` contains:

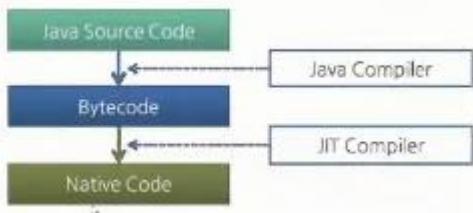
- identity hash code - от нашето Java приложение
- age
- lock record address - при синхронизация
- monitor address
- state (unlocked, light-weight locked, heavy-weight locked, marked for GC GarbageCollection)
- biased / biasable (includes other fields such as thread ID) - пристрастен - скъпо е една нишка да я изпълняваме на един процесор, после да я изпълняваме на друг процесор в друг етап от изпълнението й. За тази цел JVM може да зададе дадена нишка да се изпълнява само на конкретно ядро от процесора! Т.е. не се случва т.нр. Context Switching! И това подобрява бързодействието!

3. Execution engine

```
while(true) {
    bytecode b = bytecodeStream[pc++];
    switch(b) {
        case iconst_1: push(1); break;
        case iload_0: push(local(0)); break;
        case iadd: push(pop() + pop()); break;
    }
}
```

Different execution techniques:

- Interpreting - **вземаме следващата bytecode процесорна инструкция и я изпълняваме**
- Just-in-time (JIT) compilation - **вземаме native code процесорна инструкция и я изпълняваме**
- Adaptive optimization (determines “hot spots” by monitoring execution) - преценява дали се компилира и изпълнява впоследствие на bytecode или на native code



JIT compilation

- Triggered asynchronously by counter overflow for a method/loop (interpreted counts method entries and loopback branches) - всеки път проверява колко пъти би се извикал дадения метод
- Produces generated **native code (машинен код)** and relocation info (transferred on next method entry)
- In case JIT-compiled code calls not-yet-JIT-compiled code, then control is transferred to the interpreter!
- когато JIT-compiled code (**native машинен код**) метод извиква друг метод от нашето приложение, който е само интерпретиран, то JIT компилатора се грижи кой метод как да го изпълни и дали да не изпълни текущия метод на принципа на **On-Stack Replacement** (да го върне текущия метод да му се изпълни байткода вместо native машинния код)
- Compiled code may be forced back into interpreted bytecode (deoptimization)
- Is complemented by On-Stack Replacement - turn dynamically interpreted to JIT compiled code and vice-versa - dynamic optimization/deoptimization
- JIT compiler is more optimized for server VM - the so called server mode - hits big start-up time compared to client VM, but executes faster the logic afterwards.

JIT compilation flow

To native machine code during normal bytecode execution:

- Bytecode is turned **into a graph**
- The graph is turned **into a linear sequence** of operations that manipulate an infinite loop of virtual registers (each node places its result in a **virtual register**)
- Physical registers** are allocated for virtual registers (the program stack might be used in case virtual registers exceed physical registers)
- Native machine code** for each operation is generated using its allocated registers

Tiered Compilation in JVM

Client and server - C1 and C2 in JVM. Now they work together and not separately

<https://www.baeldung.com/jvm-tiered-compilation>

A JIT compiler **compiles bytecode to native code for frequently executed sections**.

The **client compiler**, also called **C1**, is a type of a **JIT compiler optimized for faster start-up time**. It tries to optimize and compile the code as soon as possible.

The **server compiler**, also called **C2**, is a type of a **JIT compiler optimized for better overall performance**. C2 observes and analyzes the code over a longer period of time compared to C1. This allows C2 to make better optimizations in the compiled code.

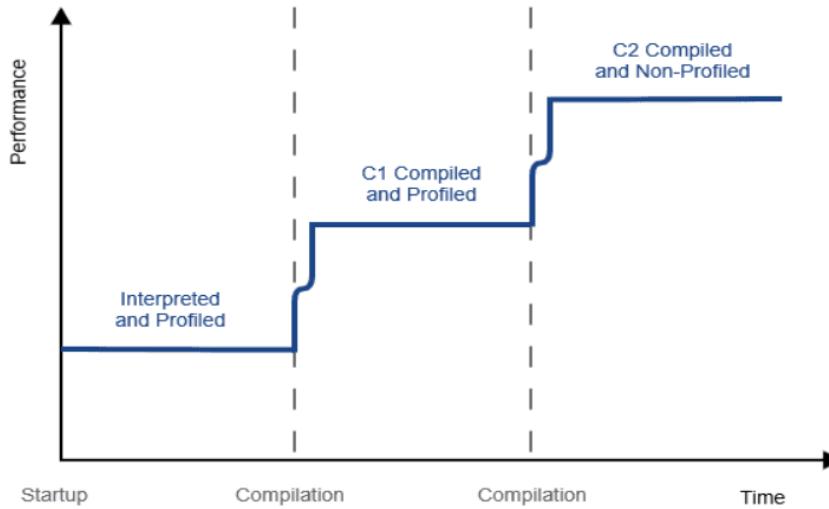
The C2 compiler often takes more time and consumes more memory to compile the same methods. However, it generates better-optimized native code than that produced by C1.

The tiered compilation concept was first introduced in Java 7. Its goal was to **use a mix of C1 and C2 compilers in order to achieve both fast startup and good long-term performance**.

Compilation Levels when tiered compilation enabled

Tiered compilation is **enabled by default since Java 8**. It's highly recommended to use it unless there's a strong reason to disable it.

Even though the JVM works with only **one interpreter** and **two JIT compilers**, there are **five possible levels of compilation**. The reason behind this is that the C1 compiler can operate on three different levels. The difference between those three levels is in the amount of profiling done.



Level 0 – Interpreted Code

Initially, JVM interprets all Java code. During this initial phase, the performance is usually not as good compared to compiled languages.

However, the JIT compiler kicks in after the warmup phase and compiles the hot code at runtime. The JIT compiler makes use of the profiling information collected on this level to perform optimizations.

Level 1 – Simple C1 Compiled Code

On this level, the JVM compiles the code using the C1 compiler, but without collecting any profiling information. The JVM uses level 1 for **methods that are considered trivial**.

Due to low method complexity, the C2 compilation wouldn't make it faster. Thus, the JVM concludes that there is no point in collecting profiling information for code that cannot be optimized further.

Level 2 – Limited C1 Compiled Code

On level 2, the JVM compiles the code using the C1 compiler with light profiling. The JVM uses this level **when the C2 queue is full**. The goal is to compile the code as soon as possible to improve performance.

Later, the JVM recompiles the code on level 3, using full profiling. Finally, once the C2 queue is less busy, the JVM recompiles it on level 4.

Level 3 – Full C1 Compiled Code

On level 3, the JVM compiles the code using the C1 compiler with full profiling. Level 3 is part of the default compilation path. Thus, the JVM uses it in **all cases except for trivial methods or when compiler queues are full**.

The most common scenario in JIT compilation is that the interpreted code jumps directly from level 0 to level 3.

Level 4 – C2 Compiled Code

On this level, the JVM compiles the code using the C2 compiler for maximum long-term performance. Level 4 is also a part of the default compilation path. The JVM uses this level to **compile all methods except trivial ones**.

Given that level 4 code is considered fully optimized, the JVM stops collecting profiling information. However, it may decide to deoptimize the code and send it back to level 0.

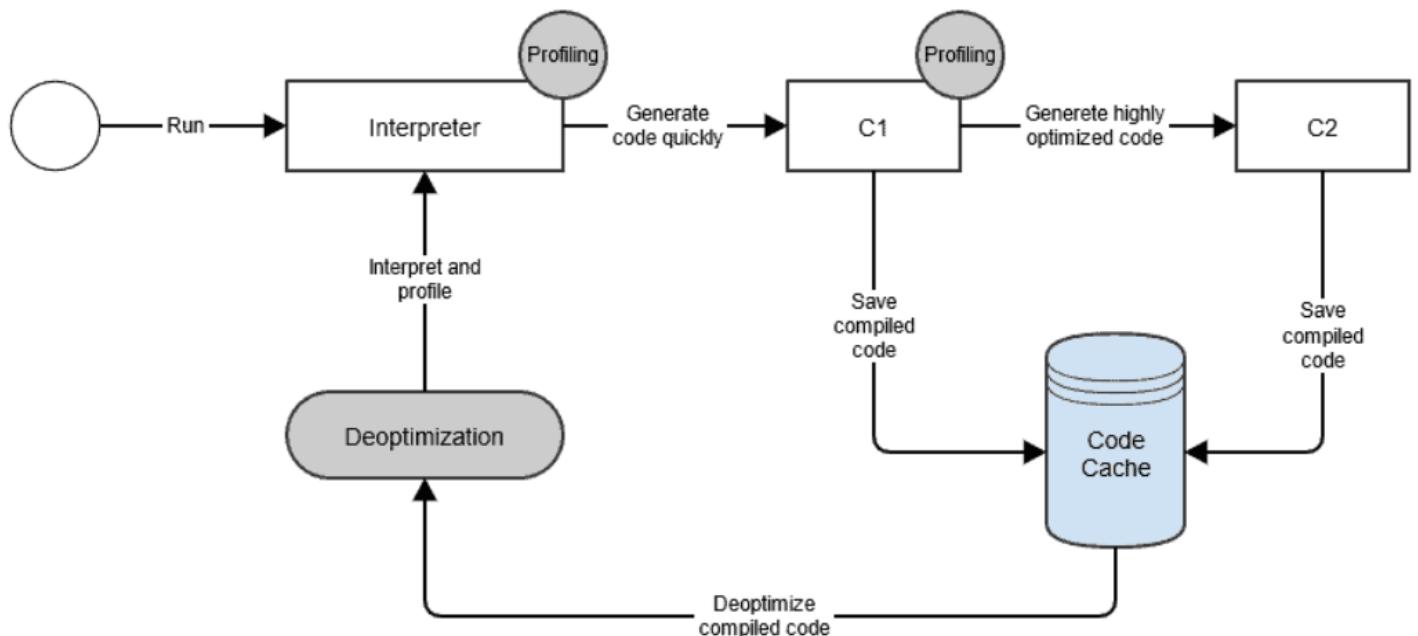
Thresholds for Levels and Method Compilation

In order to check the default thresholds used on a specific Java version, we can run Java using the `-XX:+PrintFlagsFinal` flag:

```
java -XX:+PrintFlagsFinal -version | grep CompileThreshold
intx CompileThreshold = 10000
intx Tier2CompileThreshold = 0
intx Tier3CompileThreshold = 2000
intx Tier4CompileThreshold = 15000
```

We should note that the **JVM doesn't use the generic `CompileThreshold` parameter when tiered compilation is enabled**.

In summary, the JVM initially interprets a method until its invocations reach the `Tier3CompileThreshold`. Then, it **compiles the method using the C1 compiler while profiling information continues to be collected**. Finally, the JVM compiles the method using the C2 compiler when its invocations reach the `Tier4CompileThreshold`. Eventually, the JVM may decide to deoptimize the C2 compiled code. That means that the complete process will repeat.



Typical execution flow

When using the `java/javaw` launcher:

1. Parse the command line options - **како например `-d`**
2. Establish the heap sizes and the compiler type (client or server)

3. Establish the environment variables such as **CLASSPATH** (компилираните до bytecode .java класове от нашето приложение – тоест .class видими за нас декодирани byte файлове)
4. If the java Main-Class is not specified on the command line, then fetch the Main-Class name from the JAR's manifest
5. Create the VM using **JNI_CreateJavaVM** (Java Native Interface, на C++) in a newly created thread (non primordial thread)
6. Once the VM is created and initialized, load the Main-Class
7. Invoke the **main** method in the VM using **CallStaticVoidMethod**
8. Once the **main** method completes check and clear any pending exceptions that may have occurred and also pass back the exit status

Detach the main thread using `DetachCurrentThread`, by doing so we decrement the thread count so the `DestroyJavaVM` can be called safely.

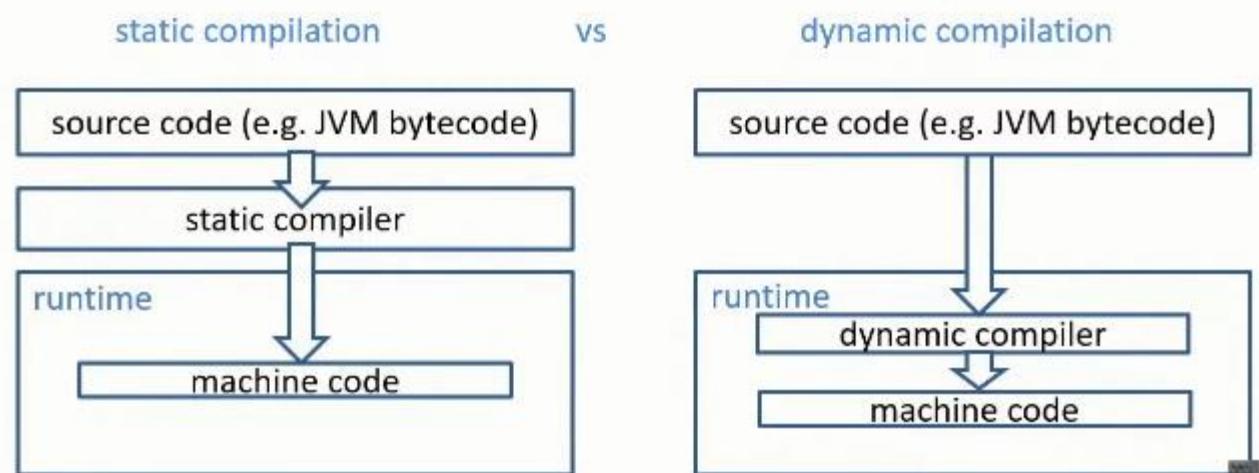
55.2. GraalVM

Some background

Статична компилация - когато нашия код **се компилира до bytecode** със **javac** компилатора/командата. И по време на изпълнение средата(JVM например) изпълнява bytecode-а **или** машинния native code.

Динамична компилация - например това го прави JIT компилатора - **изпълнява само машинен код!**

- The JVM performs **static compilation** of Java sources to **bytecode**
- The JVM may perform **dynamic compilation** of bytecode (**from bytecode**) to **machine code** for various optimizations using a JIT (Just-in-Time) compiler



Essentially, **static** linking involves compiling libraries into your app or program **as part of the build process**.

Dynamic linking lets the operating system hold off and load shared libraries into memory only when the app is launched.

Info

- GraalVM is a new JIT compiler for the JVM
- **Brings the performance of Java to scripting languages (via the Truffle API)** - възможността да се използват предимствата на Java за по-бързо интерпретиране/изпълнение на компилирания код в сравнение ако просто даден браузър само интерпретира дадено JavaScript приложение/web страница
- Written in Java
- GraalVM е нещо средно между клиентски и сървърен JIT компилатор (при стартиране прави повече оптимизации/опреации в сравнение с клиентския JIT, но все пак по-малко

оптимизации/операции в сравнение със сървърния JIT), но усъвършенстван да оптимизира още повече.

- Предназначен за скриптови езици като JS, Ruby
- Под капака GraalVM за някои оптимизации/операции си взаимодейства и с HotSpot JVM

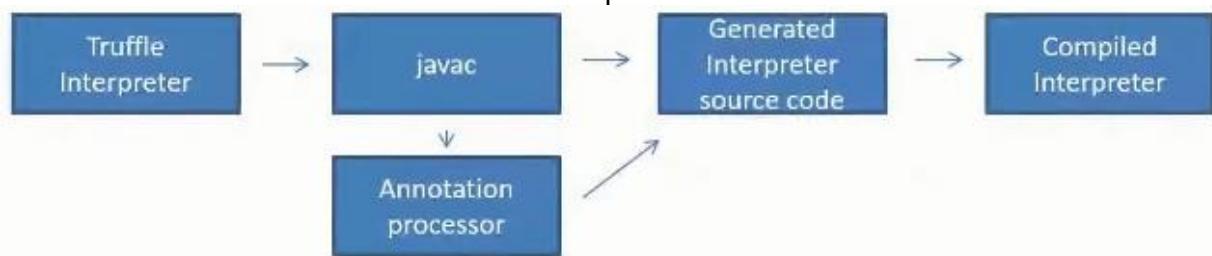
- In essence, the Graal JIT compiler generates machine code from an optimized **AST(Abstract syntax tree)** rather than bytecode
- However, the Graal VM has both **AST(Abstract syntax tree)** and bytecode interpreters

- The Graal VM supports the following types of compilers:

```
-vm server-nograal //server compiler  
-vm server //server compiler using Graal  
-vm graal //Graal compiler using Graal  
-vm client-nograal //client compiler  
-vm client //client compiler running Graal
```

The Truffle API

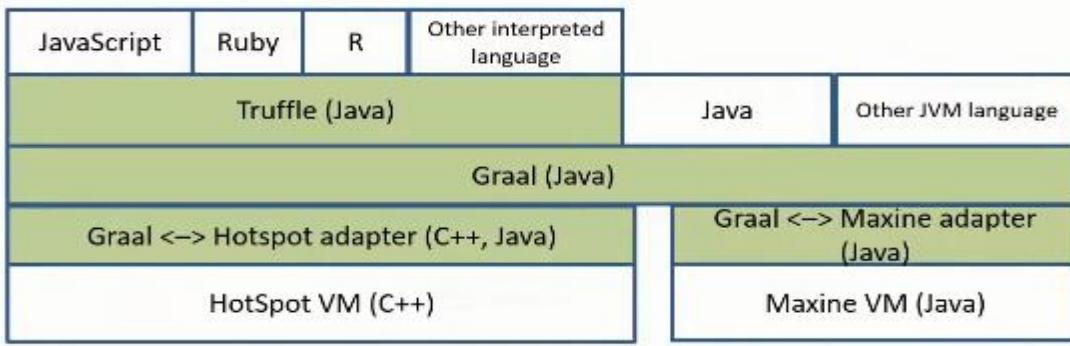
- is a Java API
- Provides AST (Abstract Syntax Tree) representation of source code
- Самото АПИ предоставя възможност за реализация на имплементация на даден програмен език, която имплементация да може да се изпълни от/на GraalVM. За целта различни елементи от нашия интерпретиран език се представят в рамките на една структура (предоставена от Truffle API-то) и тази структура се използва от GraalVM за да се генерира т.нар. **AST (Abstract Syntax Tree)**
- Provides a mechanism to convert the generated AST into a **Graal IR (intermediate representation)**
- Възможно е и от bytecode-а да стигнем до **Graal IR (intermediate representation)**
- The Truffle API is declarative (uses Java annotations)
- The AST graph generated by Truffle is a mixture of **control flow**(условни структури, for цикли, и т.н.) and **data flow graph** (местата през които да преминат данните в рамките на нашето приложение)
- Essential feature of the Truffle API is the ability to specify node specializations used in node rewriting
- The Truffle API is used in conjunction with custom annotation processor that generates code based on the Truffle annotations used in the interpreter classes.



Ruby & RubyJ пример - бекенд зарежда в пъти по-бързо през TruffleApi-то на RubyJ

А какво прави Java компилацията до bytecode и евентуално до native machine code? - като види цикъл с 5 и го прави на 5 метода. И по-бързо се изпълнява. Та и затова като мине през TruffleApi и е в пъти по-бързо отколкото ако се интерпретира. **Но тук говорим за бекенда на дадения скриптов език.**

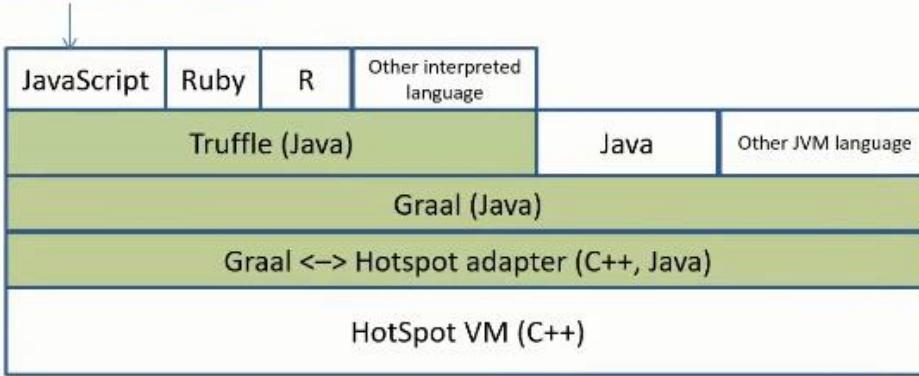
General structure of GraalVM



Let's see, for example, how a JavaScript interpreter works in Graal ...

1. Run JavaScript file: app.js
2. Подаваме app.js за изпълнения от GraalVM - JavaScript Interpreter parses app.js and converts it to **Truffle AST (Abstract Syntax Tree)**. Трябва да имаме интерпретатор (който сме написали), който да парсне този app.js файл до AST - графова структура
3. The Thruffle AST is converted into a new graph - т.нар. **Graal IR AST (intermediate representation)**
4. The Graal VM has bytecode and AST interpreters along with a standard JIT compiler from HotSpot (optimizations and AST lowering is performed **to generate machine code** and perform partial evaluation)
5. When parts of app.js are compiled to machine code by the Graal compiler, then the compiled code is transferred to HotSpot for execution by means of HotSpot APIs and with the help of a **Graal-HotSpot adapter interface**
6. Compiled code can also be deoptimized and **control is transferred back to the interpreter** (from HotSpot VM back to Graal VM - e.g. when an exception occurs, an assumption fails or a guard is reached)

Run JavaScript file: app.js



Optimizations done on the Graal IR (intermediate representation) include:

- Method inlining
- Partial escape analysis
- Inline caching
- Constant folding
- Arithmetic optimizations and others...

Реално такъв вид оптимизации могат да бъдат направени и от стандартното HotSpot JVM!

Currently supported languages from Graal VM include:

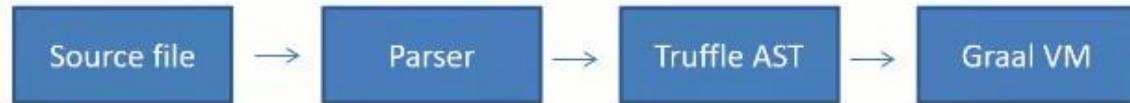
- JavaScript - (GraalJS)
- Език R - за статистически изчисления се използва най-вече - (FastR)
- Ruby - (RubyTruffle)
- Experimental interpreters for C, Python, Smalltalk, LLVM IR and others

- Graal.JS - shows improvement of 1.5x in some cases with a peak of 2.6x compared to V8 (running Google Octane's benchmark) - <https://github.com/oracle/graaljs>
- RubyTruffle - shows improvements between 1.5x and 4.5x in some cases with a peak of 14x compared to JRuby
- FastR - shows improvements between 2x and 39x in some cases with a peak of 94x compared to GnuR

SimpleLanguage project

<https://github.com/graalvm/simplelanguage>

The SimpleLanguage project provides a showcase on how to use the Truffle APIs



55.3. Garbage collection

Info

- Garbage collection is the process of scanning heap memory for unused objects and cleaning them thus reclaiming memory
- Garbage collection in the JVM is implemented by means of special modules called garbage collectors
- Garbage collectors are optimized to work over different heap areas (**young and old generations**)
- Whenever the young generations (minor collections) fills up or the old generation (major collections) need to be cleaned a “**Stop The World**” event appears
- In a “Stop The World” event application threads are paused during garbage collection
- In that regard garbage collection might be disruptive for performance in some high-frequency applications, i.e. in fintech. **Затова за такива приложения даже се елиминира изцяло процеса по garbage collection-а!**

Обекти, за които JVM прецени че живеят по-дълго от очакваното се местят от Young generation към Old / Tenured Generation!

Обектите от Young generation се изчистват по-лесно и бързо от garbage collector-а, без да се прави задълбочен анализ от garbage collector-а.

Чистенето на обекти от Old / Tenured Generation става чрез паузиране, което за някои приложения е ключово. Затова трябва да се стремим чистенето на обекти от Old / Tenured Generation да бъде минимално по дължина време!!!

Types of garbage collectors

Different types of Java collectors exist at present and **it is an area of active research and development**:

-XX:-UserSerialGC - в допълнение към основната терминална команда когато стартираме приложението - можем да зададем какъв garbage collector да използва JVM.

- **Serial GC:** garbage collection is done **sequentially**

```
-XX:-UseSerialGC
```

- **Parallel GC:** uses **multiple threads** for garbage collection (default in JDK 7 and 8):

```
-XX:-UseParallelGC
```

```
-XX:-UseParallelOldGC
```

- **Concurrent Mark Sweep (CMS):** works concurrently along the application threads trying to minimize pauses - самият garbage collector ползва няколко нишки и отделно от това тези нишки работят паралелно върху/по различните thread-ове/нишки от нашето приложение.

```
-XX:-UseConcMarkSweepGC
```

- **G1:** available as of JDK 7, parallel, concurrent and incrementally compacting low-pause collector (**default in JDK 9 and later**)

```
-XX:+UseG1GC
```

- **Epsilon GC:** this is a no-op/null garbage collector introduced in JDK 11 that allocates memory but does not do garbage collection - използва се например при тези fintech приложения където все пак трябва да се зададе като настройка тип колектор, и в случая колектора прави всичко останало, но без реалното събиране на боклука от паметта!!!!

```
-XX:+UnlockExperimentalVMOptions
```

```
-XX:+UseEpsilonGC
```

- **ZGC:** also introduced in JDK 11, still experimental, tries to reduce pause times in large scale Java applications

```
-XX:+UnlockExperimentalVMOptions
```

```
-XX:+UseZGC
```

- **Shenandoah GC:** available in JDK 13 and later (also in some earlier JDK versions), aims to reduce pause times by working concurrently with the application

```
-XX:+UnlockExperimentalVMOptions
```

```
-XX:+UseShenandoahGC
```

Като цяло, всеки по-нов garbage collector цели да се справя по-добре в случаите когато има а “Stop The World” event.

56. Jenkins

- Jenkins is an open-source automation server that helps automate various tasks in software development and delivery processes
- It provides a robust and extensible platform for continuous integration (CI) and continuous delivery (CD)
- Решава същата задача като GitHub actions, CI & CD на GitLab, на GitBucket

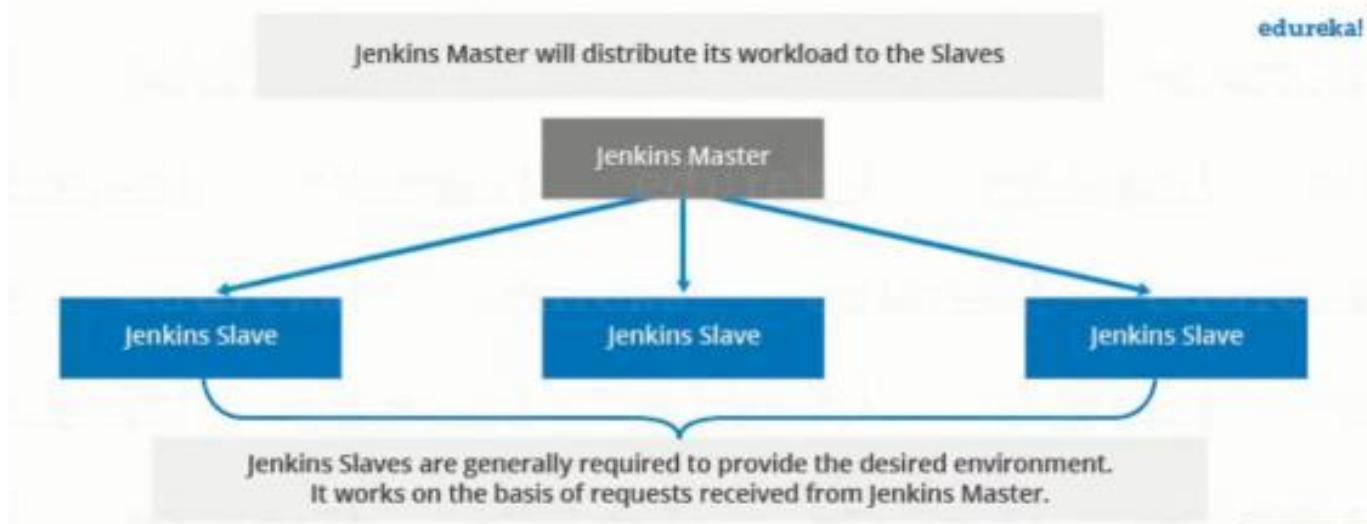
Key features

- **Continuous Integration (CI):** Jenkins enables developers to integrate code changes regularly, ensuring early detection of issues and improving collaboration
- **Continuous Delivery (CD):** Jenkins facilitates the automated deployment of applications to various environments, streamlining the release process
- **Extensibility:** Jenkins offers a vast ecosystem of plugins that extend its functionality, allowing integration with different tools and technologies.
- **Distributed Architecture:** Jenkins supports distributed builds, allowing for efficient resource utilization across multiple machines or agents - може на повече от една машина да се изпълняват тези pipelines (ако една компания има 500 человека, то не може един човек да чака 4 часа да му мине билда!)

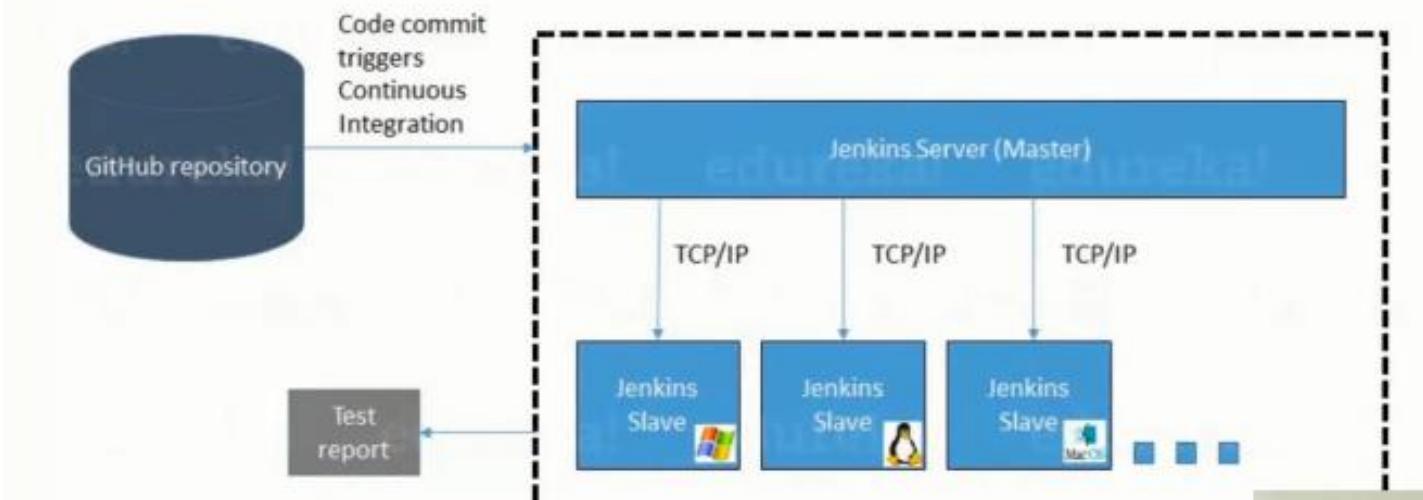
Use cases

- Continuous Integration and Testing: Jenkins can build, test and validate code changes automatically, ensuring a stable and reliable software product
- Continuous Delivery and Deployment: Jenkins enables the automated deployment of applications to multiple environments such as development, staging and production
- Task Automation: Jenkins can automate various repetitive tasks such as generating reports, sending notifications, and scheduling jobs.

How Jenkins work (Master - Slave)



How Jenkins Master and Slave Architecture works?



В случая когато ще използваме Java програмен език, то JVM машината веднъж инсталирана/конфигурирана тя работи на всяка операционна система.

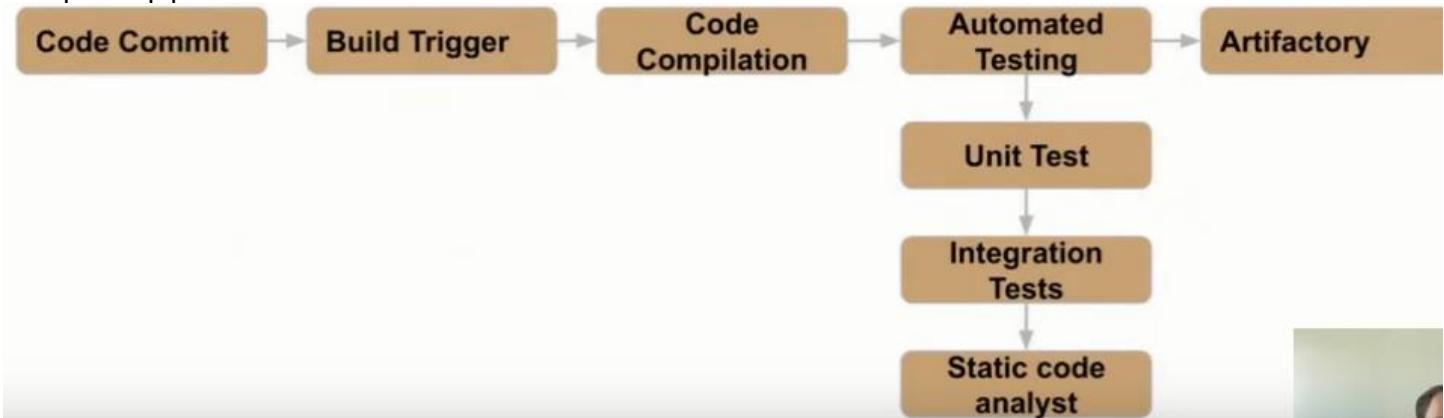
Обаче ако проекта ни е с JS/ReactJS, то има в по-голяма степен значение slave-а ни на коя ОС е нагласен!

CI & CD

What is Continuous Integration (CI)?

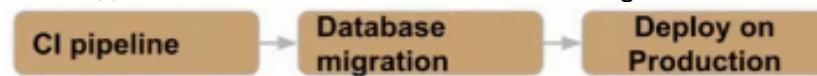
- Continuous Integration **pipeline** is a development practice where developers frequently integrate their code changes into a shared repository
- The integration process is automated and accompanied by various checks and validations
- Could be executed on 1 or more Jenkins slave nodes simultaneously

Sample CI pipeline:



What is Continuous Delivery (CD)?

- Continuous Delivery is a software development approach that focuses on automating and streamlining the software release and deployment processes
- It aims to deliver software changes more frequently, reliably, and with reduced manual effort
- Надгражда CI pipeline-а като добавя някои от следните неща: миграция на база данни, деплоиване на съответния docker image на съответната среда



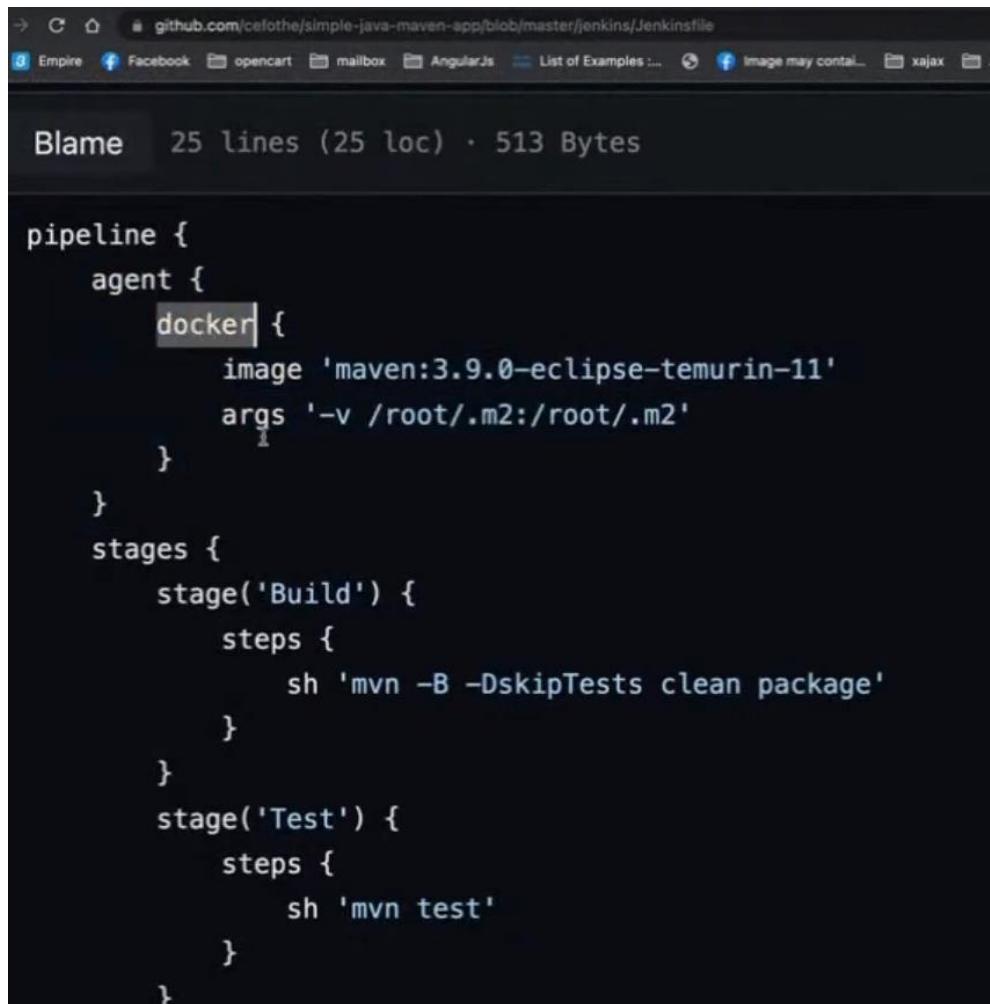
How Jenkins file will look like

Sample of a Jenkins file

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

agent Java, Maven, etc.



The screenshot shows a GitHub browser interface with the URL `github.com/celofthe/simple-java-maven-app/blob/master/Jenkinsfile`. The page displays a Jenkins pipeline configuration. At the top, it says "Blame 25 lines (25 loc) · 513 Bytes". The Jenkinsfile content is shown below:

```
pipeline {
    agent {
        docker {
            image 'maven:3.9.0-eclipse-temurin-11'
            args '-v /root/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
    }
}
```

```
stage('Deliver') {
    steps {
        sh './jenkins/scripts/deliver.sh'
    }
}
```

Installing Jenkins java app with maven

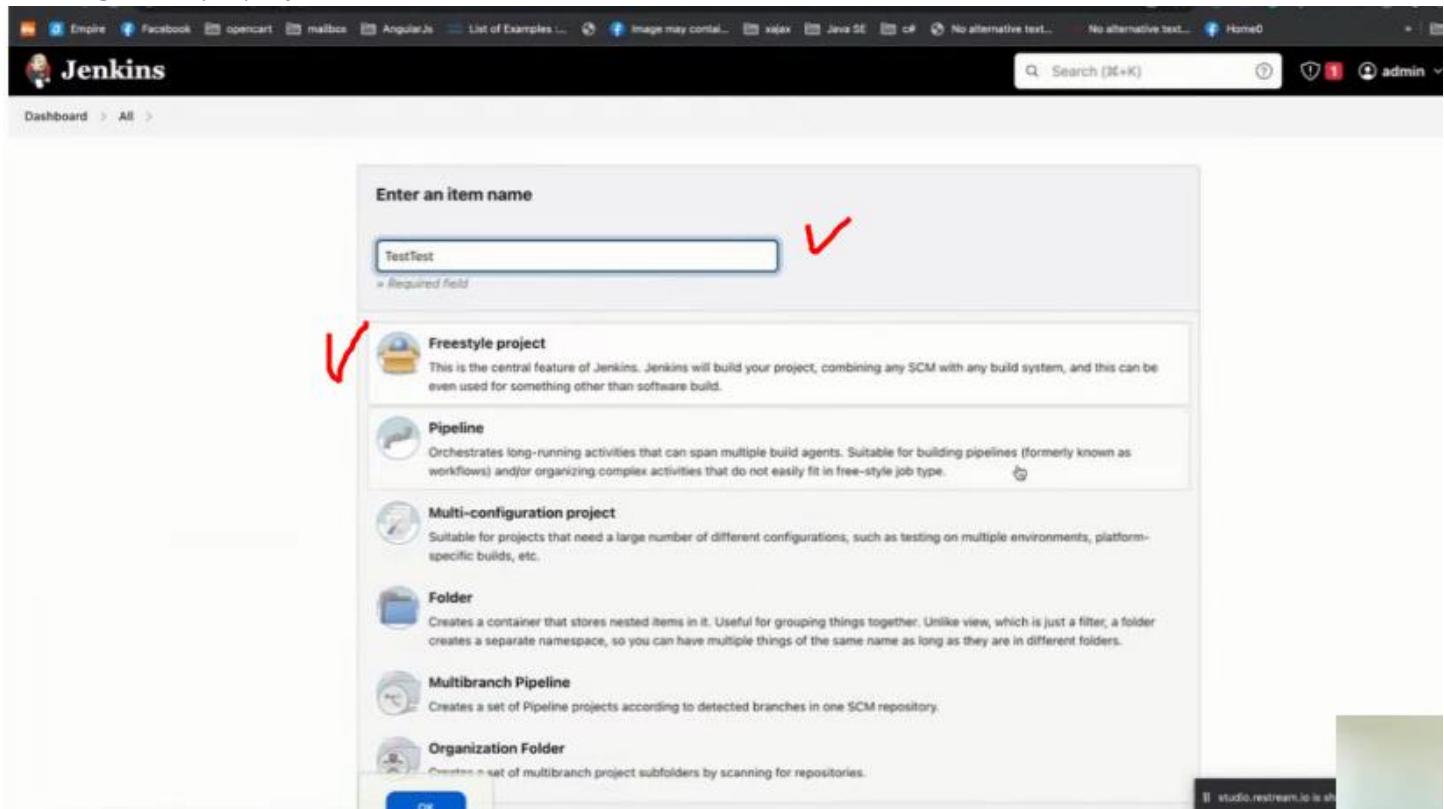
How to run Jenkins locally:

<https://www.jenkins.io/doc/tutorials/build-a-java-app-with-maven/>

Тази инсталация посочена в линка работи в docker контейнер, като преди да започнем да работим с този контейнер, то имаме инсталиран в него JDK and Maven.

В други записи има примери как това става даже по-лесно ако работим с JS.

Creating Freestyle project



localhost:8080/job/BGJUGFirstPipeline/configure

Dashboard > BGJUGFirstPipeline > Configuration

Configure

GitHub project

Project url ?

Advanced ▾

Pipeline speed/durability override ?
 Preserve stashes from completed builds ?
 This project is parameterized ?
 Throttle builds ?

Build Triggers

Build after other projects are built ?
 Build periodically ?
 GitHub hook trigger for GITScm polling ?
 Poll SCM ?
 Quiet period ?
 Trigger builds remotely (e.g., from scripts) ?

localhost:8080/job/BGJUGFirstPipeline/configure

Dashboard > BGJUGFirstPipeline > Configuration

Configure

Advanced ▾

General
 Advanced Project Options
 Pipeline

Pipeline

Definition

Pipeline script

```
Script ?  
  1  pipeline {  
  2      stages {  
  3          stage('Build') {  
  4              steps {  
  5                  echo 'Building...'  
  6              }  
  7          }  
  8      }  
  9  }  
10+  stage('Test') {  
11+      steps {  
12+          echo 'Testing...'  
13+      }  
14+  }  
15+  stage('Deploy') {  
16+      steps {  
17+          echo 'Deploying...'  
18+      }  
19+  }  
20+ }  
21 }  
22 }
```

Use Groovy Sandbox ?

Pipeline Syntax

Save Apply

Pipeline BGJUGFirstPipeline

Stage View

Average stage times:
(Average full run time: ~793ms)

Build	Test	Deploy	Clean Up
45ms	28ms	28ms	26ms
32ms	Success Logs	28ms	26ms
59ms	32ms	28ms	

Permalinks

#2
Jul 4, 2023, 10:06 AM

#1
Jul 4, 2023, 10:06 AM

Atom feed for all Atom feed for failures

Creating Multibranch pipeline

Можем да добавяме много бранчове, и даже и да включим информация за PRs.

Enter an item name

Required field

Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Pipeline

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

Multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Folder

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

Multibranch Pipeline

Creates a set of Pipeline projects according to detected branches in one SCM repository.

Organization Folder

Creates a set of multibranch project subfolders by scanning for repositories.

OK

studio.restream.io is sharing your screen. Stop sharing Hide

The screenshot shows the Jenkins Multibranch Pipeline Test dashboard. On the left, there's a sidebar with links like Status, Configure, Scan Multibranch Pipeline Now, Scan Multibranch Pipeline Log, Multibranch Pipeline Events, Delete Multibranch Pipeline, People, Build History, and Open Blue Ocean. The main area has a title 'MultyBranchPipelineTest' with a 'Disable Multibranch Pipeline' button. Below it, a 'Branches (2)' section lists 'develop' and 'master'. A table provides details for each branch: Last Success (N/A), Last Failure (N/A), Last Duration (N/A), and Fav (with a star icon). At the bottom, there are icons for S, M, L, an icon legend, and three Atom feed links: 'Atom feed for all', 'Atom feed for failures', and 'Atom feed for just latest builds'. A status message at the bottom right says 'studio.restream.io is sharing your screen. Stop sharing'.

57. Github actions

<https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>

<https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven>

<https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-gradle>

What is Git Actions

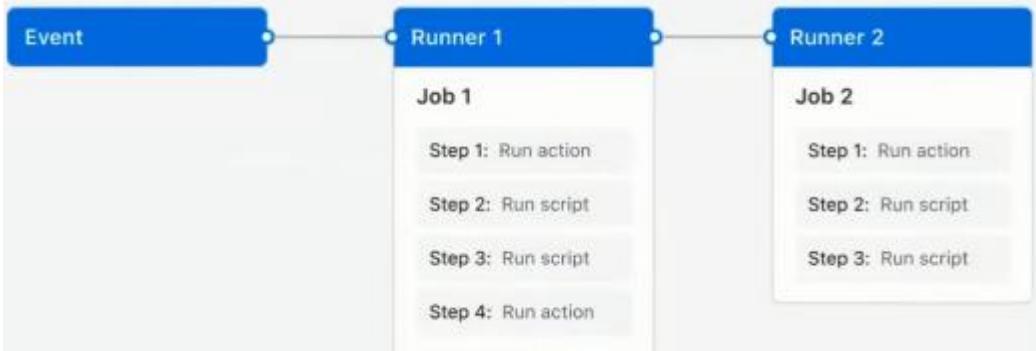
GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.

Също така може да си добавим/пълъгнем и допълнителни actions като например cron jobs.

The components of GitHub Actions

Тригърване на ивента ръчно - например при Jenkins ръчно при **build-a**. Но може да става и автоматично тригърването на ивента.

Има рънъри за 3 операционни системи (за Windows, Linux and MacOS) – като при Jenkins.



Workflows

A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked into your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.

```

  .github
    workflows
      cd-analytics.yml
      cd-backoffice-gateway.yml
  
```

Events

An event is a specific activity in a repository that triggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository.

Jobs

A job is a set of steps in a workflow that is executed on the same runner. Each step is either a shell script that will be executed, or an action that will be run. Steps are executed in order and are dependent on each other. Since each step is executed on the same runner, you can share data from one step to another. For example, you can have a step that builds your application followed by a step that tests the application that was built.

Actions

There are ready pre-built actions in GitHub. They are similar to the Jenkins plugins.

An action is a custom application for the GitHub Actions platform that performs a complex but frequently repeated task. Use an action to help reduce the amount of repetitive code that you write in your workflow files. An action can pull your git repository from GitHub, set up the correct toolchain for your build environment, or set up the authentication to your cloud provider.

Runners

A runner is a server that runs your workflows when they are triggered. Each runner can run a single job at a time. GitHub provides Ubuntu Linux, Microsoft Windows, and MacOS runners to run your workflows. Each workflow run executes in a fresh, newly-provisioned virtual machine.

Пример за сі или cd .yml файл:

```
build-java-with-maven.yml
name: Build with Java and Maven
run-name: Java with maven
#on: [push]
on:
  push:
    branches:
      - 'develop'

  pull_request:
    branches:
      - 'master'

jobs:
  build-with-maven:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Starting job"
      - name: Checkout repository
        uses: actions/checkout@v3

      - uses: actions/setup-java@v3
        with:
          java-version: '11'
          distribution: 'temurin'
          cache: maven

      - name: Build with maven
        run: mvn clean install
```

58. HTTP with java.net library

HTTP programmatically with java.net - the GET request simple example:

```
public class Main {
    public static void main(String[] args) throws IOException {
        String test = getArticle("JDK");
        System.out.println(test.contains("Java"));
    }

    public static String getArticle(String title) {
        try {
            java.net.URL url = new
java.net.URL("https://en.wikipedia.org/api/rest_v1/page/html/%s".formatted(title));
            java.net.HttpURLConnection con = (java.net.HttpURLConnection)
url.openConnection();
            // Setting the request method and properties.
            con.setRequestMethod("GET");

            int responseCode = con.getResponseCode();

            if (responseCode == java.net.HttpURLConnection.HTTP_OK) { // success
                java.io.BufferedReader in = new java.io.BufferedReader(new
java.io.InputStreamReader(con.getInputStream()));
                String inputLine;
                java.lang.StringBuffer responseText = new java.lang.StringBuffer();

                while ((inputLine = in.readLine()) != null) {
                    responseText.append(inputLine);
                }
                in.close();

                // print result
                return responseText.toString();
            } else {
                return "GET request did not work.";
            }
        } catch (java.io.IOException e){
            return "IO exception occurred";
        }
    }
}
```

59. Websocket in java.net.Socket

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.Socket;

public class Client {
    public static void main(String[] args) throws IOException {
        Socket socket = null;
        BufferedWriter writer = null;

        try {
            socket = new Socket("localhost", 9000);
            writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
            writer.write("message1\n");
            writer.write("message2\n");
        }
```

```

writer.write("stop\n");
writer.flush(); //за да изпратим съобщението към сървъра

        BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        String message = reader.readLine();
        System.out.println(message);
        writer.close();
        reader.close();
    } finally {
        if (socket != null) {
            socket.close();
        }
    }
}

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = null;
        try {
            Executor threadPool = Executors.newFixedThreadPool(100);
            socket = new ServerSocket(9000);

            while (true) { //the so-called busy wait server
                final Socket clientSocket = socket.accept();
                threadPool.execute(() -> {
                    BufferedReader reader = null;
                    BufferedWriter writer = null;
                    try {
                        reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
                        do {
                            String message = reader.readLine();
                            System.out.println(message);
                            if ("stop".equals(message)) {
                                writer = new BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream()));
                                writer.write("Hey Client - request finished!\n");
                                writer.flush();
                                break;
                            }
                        } while (true);
                    } catch (IOException ex) {
                        ex.printStackTrace();
                    } finally {
                        try {
                            reader.close();
                            writer.close();
                            clientSocket.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                });
            }
        } finally {
    }
}

```

```

        if (socket != null) {
            socket.close();
        }
    }
}

```

59. Project Amber

Project **Amber** is about Record Patterns - <https://openjdk.org/jeps/440>

Record patterns and pattern matching. The key to understanding these related terms is first understanding what a pattern is. Then we can combine the concept of patterns to **record types**, and finally go deeper into pattern matching with **instanceof** and **switch expressions**.

Goals

Extend pattern matching to destructure instances of `record` classes, enabling more sophisticated data queries.

Add nested patterns, enabling more composable data queries.

```

// Prior to Java 16
if (obj instanceof String) {
    String s = (String) obj;
    ... use s ...
}

// As of Java 16
if (obj instanceof String s) {
    ... use s ...
}

```

Pattern matching and records

```

// As of Java 16
record Point(int x, int y) {}

static void printSum(Object obj) {
    if (obj instanceof Point p) {
        int x = p.x();
        int y = p.y();
        System.out.println(x+y);
    }
}

// As of Java 21
static void printSum(Object obj) {
    if (obj instanceof Point(int x, int y)) {
        System.out.println(x+y);
    }
}

```

Nested

Switch expressions

60. Architecture and microservices

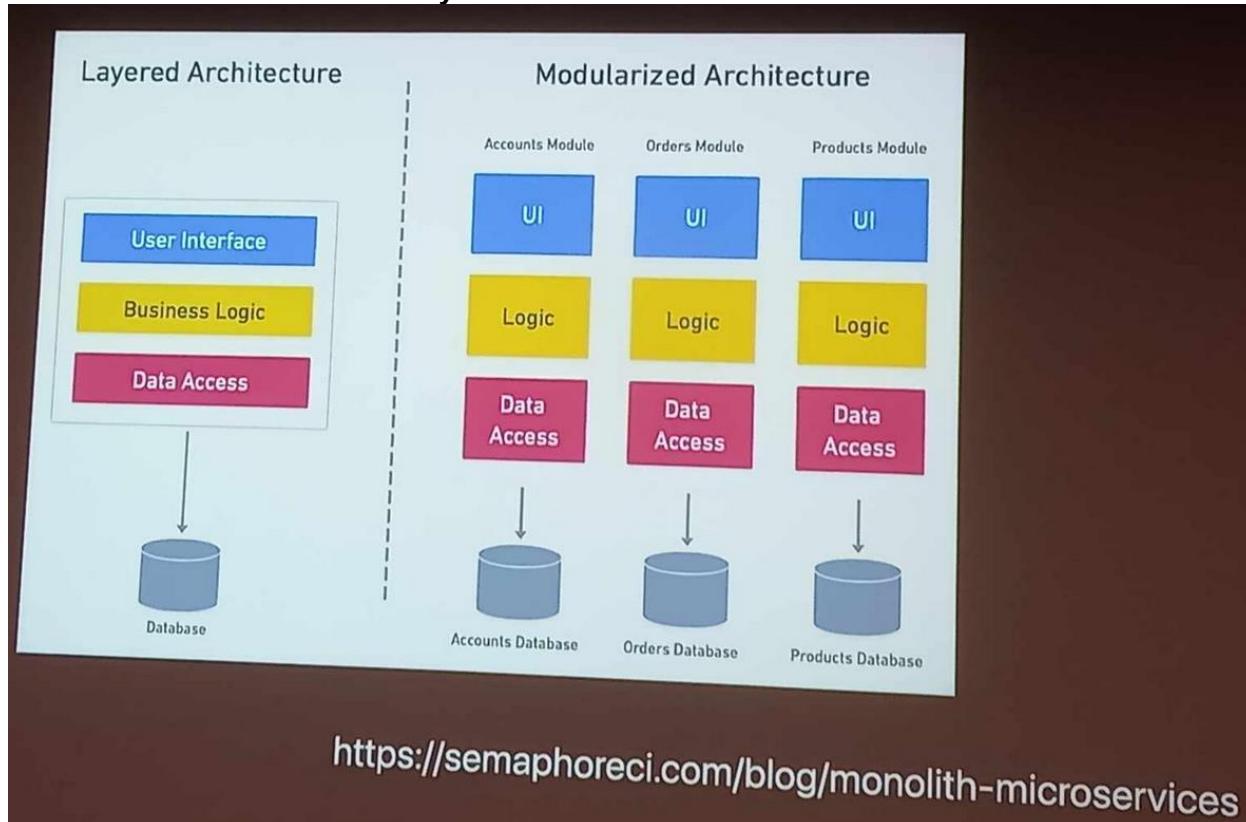
Cohesion - same matter at one place in the code

Coupling - interacting one another

Loose coupling and high cohesion make it a microservice.
High coupling makes it a monolith.

Eventual consistency - I ask the bank do I have money, and the bank answers don't know, maybe yes, maybe no can buy that thing. Who knows?

Modularized architecture vs Layered architecture



Modular Monolith Structure

bg.jug.sample

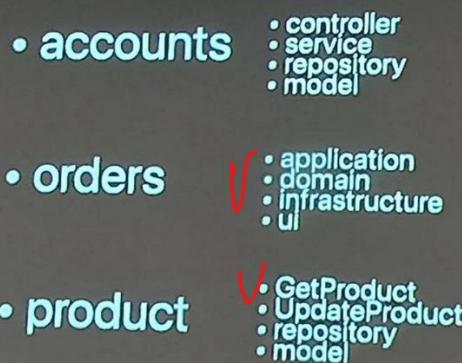
- **accounts**
 - controller
 - service
 - repository
 - model

- **orders**
 - controller
 - service
 - repository
 - model

- **product**
 - controller
 - service
 - repository
 - model

Modular Monolith Structure

bg.jug.sample



Rober "Uncle Bob" Martin: **Your architecture should tell readers about the system, not about the frameworks you used in your system!**

Screaming architecture - to scream what it is

XX. други

```
stack.stream()
    .sorted(Comparator.reverseOrder())
    .map(obj -> String.valueOf(obj))
    .collect(Collectors.joining(", "));
```

Обикновено когато имаш master обект, slave обектите не връщат мастър обекта, а реферират само към ИД-то му. За да няма cross-reference. Ако се наложи от аптеката да вади верига, ще вземе ид-то и ще извика ресурса за верига.

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
```