Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

ביטויי למבדא

מה הוא למעשה ביטוי למבדא?

- ביטויי Lambda משמשים בעיקר כדי להגדיר יישום מוטבע של ממשק פונקציונלי, כלומר, ממשק עם שיטה אחת בלבד.
- ביטוי Lambda מבטל את הצורך במחלקה אנונימית ומעניק יכולת תכנות פונקציונלית פשוטה אך עוצמתית ל-Java.
- באמצעות ביטוי למבדה, ניתן להתייחס לכל משתנה סופי או משתנה אשר מאותחל פעם אחת בלבד. ביטוי Lambda זורק שגיאת קומפילציה, אם למשתנה מוקצה ערך בפעם השנייה.

```
parameter -> expression

(parameter1, parameter2) -> expression

(parameter1, parameter2) -> { code block }
```

נבנה ממשק אשר יהיה משותף לכל הפעולות:

```
//tells us what we can use
interface MathOperation {
    int operation(int a, int b);
}
```

כעת נבנה את ביטויי הלמדא השונים:

```
//with type declaration
MathOperation addition = (int a, int b) -> a + b;

//with out type declaration
MathOperation subtraction = (a, b) -> a - b;

//with return statement along with curly braces
MathOperation multiplication = (int a, int b) -> {
    //some logic code , goes here

    //remember to return something :)
    return a * b;
};

//without return statement and without curly braces
MathOperation division = (int a, int b) -> a / b;
```

ולבסוף על מנת להפעיל את הפעולות, פשוט נקרא לביטוי הרלוונטי:

```
System.out.println("10 + 5 = "+addition.operation(10,5));
System.out.println("10 - 5 = "+subtraction.operation(10, 5));
System.out.println("10 x 5 = "+multiplication.operation(10, 5));
System.out.println("10 / 5 = "+division.operation(10, 5));
```

ופה אנחנו נכנסים למעשה לעולם של functional programming עם כל היכולות המטורפות שלו.

Functions

אז איך כתבנו קוד עד עכשיו ???

נבנה מחלקה המייצגת אדם בשם Person

```
static class Person {
    private final String name;
    private final Gender gender;

    Person(String name, Gender gender) {
        this.name = name;
        this.gender = gender;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", gender=" + gender +
                '}';
    }
}
```

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

נצור enum פשוט המייצג בן או בת

```
enum Gender {
    MALE, FEMALE
}
```

כאשר יש לנו עכשיו את מבנה המחלקה, נוסיף מידע על לתת נפח שאיתו נעבוד.

```
List<Person> people = List.of(
        new Person("Zeev", MALE),
        new Person("Amital", FEMALE),
        new Person("Hilda", FEMALE),
        new Person("Sasson", MALE),
        new Person("Michal", FEMALE)
);
```

עד היום עבדנו בצורה שנקראת Imperative approach, כלומר היינו צריך לדאוג למימוש של המחלקה, פקודות לוגיות, והתייחסות לכל מופע של המחלקה

לדוגמא, במידע שיש למעלה, היינו רוצים להדפיס רק את הבנות שנמצאות באוסף, או ליצור אוסף חדש שמכיל אך ורק בנות.
לצורך זה היינו צריכים לכתוב את הקוד הבא:

```
System.out.println("// Imperative approach");
// Imperative approach

List<Person> females = new ArrayList<>();

for (Person person : people) {
    if (FEMALE.equals(person.gender)) {
        females.add(person);
    }
}

for (Person female : females) {
    System.out.println(female);
}
```

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

אנו רואים שיש לנו שני לולאות, הראשונה מכילה אך ורק בנות, ואילו השנייה מדפיסה את כל הבנות.
הקוד מסורבל, מכיוון שהיינו צריכים לצור אוסף חדש, לעבור על האוסף הקיים ולהוסיף לאוסף החדש רק את האובייקטים שאנו רוצים.

בגישה של Declarative approach אנחנו למעשה מקטינים את הקוד משמעותית ע״י שימוש ב Functional programing, המאפשר לנו לכתוב קוד קצר, יעיל, עם תחזוקה נמוכה, ויותר מהיר מהקוד הקודם.

```
// Declarative approach
List<Person> females2 = people.stream()
        .filter(person -> FEMALE.equals(person.gender))
        .collect(Collectors.toList());
//print the result
females2.forEach(System.out::println);
```

קוד הרבה יותר פשוט, יעיל ומובן.
שימו לב, השתמשנו פה בביטויי למבדא, המצויין כזכור ע״י השימוש בסימן <-
שורה ראשונה, יצרנו מערך חדש, ובעזרת הפקודה stream, הזרמנו למעשה את המידע מהרשימה המקורית, כלומר ג׳אווה תעבור על כל איבר, ממש כמו בלולאת for each.
בשורה שנייה ביקשנו לקבל רק את האיברים המייצגים נשים.
ואילו בשורה השלישית, ביקשנו לאסוף את כל המידע לתוך אוסף.

בשורה האחרונה, הדפסנו את כל האיברים באוסף החדש שייצרנו.

על מנת לכתוב קוד בצורה יעילה כזאת, אנו צריכים להכיר פקודות חדשות ושחקנים חדשים שייעזרו לנו לממש קוד ב - Functional programing.

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

שימוש ב – Function

על מנת להבין פונקציות, בואו נראה איך עבדנו עד היום.

נצור שיטה פשוטה

```
static int incrementByOne(int number) {
    return number + 1;
}
```

על מנת לקרוא לשיטה נשתמש בפקודה

```
// Function takes 1 argument and produces 1 result
int increment = incrementByOne(1);
System.out.println(increment);
```

מבנה פונקציה :

Function <T,T> name = <expression>

אם נרצה להשתמש בשיטה הזאת במספר מקומות בתוכנה, נצטרך לשכפל אותה (big no no) או לחילופין להשתמש בה כשיטה סטטית במחלקת עזר (כזאת שכל השיטות שלה הם סטטיות)

כאשר אנו כותבים בתצורת Declarative approach אנו לא מגדירים שיטות, אלא פונקציות שאיתם נוכל להשתמש במקומות אחרים בתוכנה

```
Function<Integer, Integer> incrementByOneFunction = number -> number + 1;
```

על מנת לקרוא לפונקציה נשתמש בפקודה

```
int increment2 = incrementByOneFunction.apply(1);
System.out.println(increment2);
```

בעזרת השיטה apply, אנו מעבירים ארגומנט אל השיטה, התוצאה שנקבל תהיה במקרה הזה 2.
אבל שימו לב, הפונקציה בנוייה בצורה של tuple, כלומר זוג נתונים, כאשר הנתונים הראשון מציין את סוג הפרמטר המתקבל, והנתון השני, את סוג הפרמטר המוחזר.

ופה אנו שמים לב לשוני, בעוד ששיטה רגילה יכולה לקבל מספר גדול של פרמטרים, פונקציה יכולה במקרה הזה לקבל רק פרמטר אחד ולהחזיר פרמטר אחד.

השימוש בביטוי למבדא הינו פשוט ואנו רואים שהרבה יותר קל לקרוא את הקוד ברגע שמתרגלים למבנה של הפונקציה.

שימוש בביטוי למבדא נפוץ היום בהרבה שפות תיכנות, השמות אולי שונים אבל מבנה הקוד נשאר אותו דבר, למעשה כאשר נדבר על frontend ונכתוב לreact בעזרת java script נראה שהמבנה חוזר על עצמו תחת שם אחר arrow function.

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

היתרון הנוסף בשימוש בפונקציות הוא שאני למעשה יכול לשרשר בין הפונקציות, מה הכוונה?
אני יכול להעביר בארגומנט פונקציה אחרת ובכך להגיע לחישוב מורכב יותר.

ניקח מקרה שאני רוצה להוסיף 1 לנתון כלשהוא, ואז להכפיל את התוצאה בעשר.

בגישה רגילה בגאווה, אני צריך למעשה לקרוא לשני השיטות אחת אחרי השנייה.

```
static int incrementByOneAndMultiply(int number, int numToMultiplyBy) {
    return (number + 1) * numToMultiplyBy;
}
```

בתכנות פונקציונלי, אני יכול למעשה להצהיר על פונקציה חדשה, הדואגת להחלפת התוצאה

נצור פונקציה המטפלת בהכפלה ב
10

```
Function<Integer, Integer> multiplyBy10Function = number -> number * 10;
```

כעת כל מה שנשאר לי זה להגיד למחשב תוסיף 1 ואח"כ תכפיל ב 10

```
Function<Integer, Integer> addBy1AndThenMultiplyBy10 =
        incrementByOneFunction.andThen(multiplyBy10Function);
System.out.println(addBy1AndThenMultiplyBy10.apply(4));
```

שמו לב, המילה andThen, למעשה מבטאת את הדיבור, תוסיף 1 **ואז** תכפיל ב 10
כלומר, אני יכול לשרשר את הפונקציות והתוצאות המוחזרות בסדר שאני מחליט עליו.

אבל מה קורה אם אני רוצה לקצר את התוכנית עוד יותר, אם רק יכולתי להוסיף עוד ארגומנט.
אז ישנה אפשרות לפונקציה לקבל 2 ארגומנטים ולהחזיר תוצאה, אבל זה המקסימום שנוכל
הפקודה נקראת BiFunction, כלומר פונקציה בינארית, פונקציה המקבלת שני ארגומנטים לחישוב.

הכתיבה תהיה אז

```
BiFunction<Integer, Integer, Integer> incrementByOneAndMultiplyBiFunction =
        (numberToIncrementByOne, numberToMultiplyBy)
                -> (numberToIncrementByOne + 1) * numberToMultiplyBy;
```

שימו לב, למרות שאנו מקבלים 2 ארגומנטים, אנו משתמשים בפונקציה שכבר כתבנו על מנת לקבל את התוצאה.

חשוב לדעת !

אנו נעדיף לפרק את האלגוריתמים שלנו ליחידות קטנות ככל האפשר
דבר שיאפשר לנו לטפל נקודתית בפעולת חישוב
מאשר להתעסק עם אלגוריתם חישוב מסובך וללכת לאיבוד.

שימוש בפונקציה החדשה תהיה

```
System.out.println(incrementByOneAndMultiplyBiFunction.apply(4, 100));
```

Predicate

טיפוס Predicate הוא טיפוס המקביל אובייקט, ומחזיר true או false בהתאם לבדיקה הנערכת בפנים.

בכתיבה רגילה היינו כותבים:

```
static boolean isPhoneNumberValid(String phoneNumber) {
    return phoneNumber.startsWith("05") && phoneNumber.length() == 11;
}
```

השיטה תבדוק האם מספר הטלפון מתחיל ב05 וכמו כן שהגודל הינו 11 תווים.

שימוש בשיטה

```
System.out.println("Without predicate");
System.out.println(isPhoneNumberValid("052-4043142"));
System.out.println(isPhoneNumberValid("050-5433874"));
System.out.println(isPhoneNumberValid("054-8374892"));
```

אנו רואים שהשיטה למעשה פשוטה, ואיננה מורכבת.

אנו יכולים להשתמש ב predicate על מנת להגיע לאותה תוצאה בדיוק.

```
Predicate<String> isPhoneNumberValidPredicate = phoneNumber ->
        phoneNumber.startsWith("05") && phoneNumber.length() == 11;
```

אנו שוב רואים שימוש בביטוי למבדא, המקבל נתון מסוג מחרוזת ובודק אם הטלפון מתחיל ב-05 וכמו כן שהאורך שלו הוא 11 תווים בדיוק.

על מנת להשתמש בpredicate

```
System.out.println("With predicate");
System.out.println(isPhoneNumberValidPredicate.test("052-4043142"));
System.out.println(isPhoneNumberValidPredicate.test("050-5433874"));
System.out.println(isPhoneNumberValidPredicate.test("054-8374892"));
```

כלומר על מנת להפעיל את הפקודה, נשתמש בשיטה המובנית test אשר מקבל ארגומנט מסוג מחרוזת.

במידה ונרצה להציג הודעה יותר הגיונית למשתמש

```
System.out.println(
        "Is phone number valid and contains number 3 = " +
        isPhoneNumberValidPredicate.and(containsNumber3).test("052-4043142")
);
```

בדומה לביטויים בוליאניים, גם predicate יודע להשתמש בביטויים בוליאנים בעזרת פקודות and/or השימוש בהם הוא למעשה פשוט, ובמקרה הזה אנו משרשרים פקודות

נבנה טיפוס חדש מסוג predicate אשר יבדוק אם מספר הטלפון מכיל את הספרה 3

```
static Predicate<String> containsNumber3 = phoneNumber ->
        phoneNumber.contains("3");
```

כעת אנו רוצים לבדוק האם הטלפון שלנו מתחיל ב 05 או מכיל את הספרה 3.
בכתיבה רגילה, היינו צריכים ביטויי בוליאני המורכבת משני שיטות המחזירות ביטוי בוליאני.

שימו לב כמה פשוט זה הפוך להיות

```
System.out.println(
        "Is phone number valid or contains number 3 = " +
                isPhoneNumberValidPredicate
                .or(containsNumber3).test("052-4043142")
);
```

קראנו למעשה לפנוקציה הראשונה ושירשרנו אותה בעזרת or לפונקציה השנייה.
בעזרת שיטת test העברנו את הנתון לבדיקה.

שימו לב לפשטות של הכתיבה, ובמידה ונרצה להוסיף בדיקות נוספות, נוכל לעשות בעזרת הוספה פשוטה של עוד פקודות, למעשה אנחנו בונים אוסף של חוקים לבדיקה בצורה קלה ומהירה.

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

Consumer

מה קורה כאשר אנו רוצים לצור פונקציה, אשר מקבלת ערכים, אבל לא מחזירה כלום (void)?

לצורך זה אנו נשתמש בטיפוס מסוג Consumer שזה בדיוק הגדרת התפקיד שלו.

לצורך דוגמא נבנה מחלקה המייצגת לקוח

```
class Customer {
    private final String customerName;
    private final String customerPhoneNumber;

    Customer(String customerName, String customerPhoneNumber) {
        this.customerName = customerName;
        this.customerPhoneNumber = customerPhoneNumber;
    }
}
```

נצור כעת אובייקט מהמחלקה שכתבתנו

```
// Normal java function
Customer zeev = new Customer("Zeev", "052-4043142");
```

כעת נבנה שיטה בצורה רגיל המברכת את הלקוח

```
void greetCustomer(Customer customer) {
    System.out.println("Hello " + customer.customerName +
            ", thanks for registering phone number "
            + customer.customerPhoneNumber);
}
```

והשימוש בה פשוט

```
greetCustomer(zeev);
```

אבל אנו יכולים גם לבנות בעזרת פונקציה שלא מחזירה ערכים, כלומר Consumer

```
Consumer<Customer> greetCustomerConsumer = customer ->
        System.out.println("Hello " + customer.customerName +
                ", thanks for registering phone number "
                + customer.customerPhoneNumber);
```

והשימוש בה בדיוק כמו בפונקציה, רק שכעת אנו לא מציינים ערך החזרה, מכיוון שאנו גם לא מצפים לקבל ערך החזרה, כלומר ביצוע פעולה מסוימת מבלי קבלת חיווי חוזר.

```
// Consumer Functional interface
greetCustomerConsumer.accept(zeev);
```

אבל מה קורה אם לא היינו רוצים לחשוף את הטלפון של המשתמש?

בכתיבה רגילה, היינו מוסיפים לפונקציה פרמטר נוסף המציין האם אנו רוצים להחצין את מספר הטלפון לעולם או להעלים אותו

```
static void greetCustomerV2(Customer customer, boolean showPhoneNumber) {
    System.out.println("Hello " + customer.customerName +
            ", thanks for registering phone number "
            + (showPhoneNumber ? customer.customerPhoneNumber:"*********"));
}
```

כעת השיטה מקבל שני ארגומנטים, הראשון הוא הלקוח, והשני מציין האם להראות את הטלפון של הלקוח או לא.
בעזרת תנאי מקוצר, פעלנו להצגה או אי הצגה של הטלפון.

קריאה לשיטה תעשה

```
greetCustomerV2(zeev, false);
```

כעת בואו נהפוך את השיטה הישנה לפונקציה ללא ערך החזרה

```
BiConsumer<Customer, Boolean> greetCustomerConsumerV2 = (customer, showPhoneNumber) ->
        System.out.println("Hello " + customer.customerName +
                ", thanks for registering phone number "
                + (showPhoneNumber ? customer.customerPhoneNumber : "*********"));
```

אנו רואים שהתוכן מתנהג אותו דבר.

כלומר הפעם קיבלנו שני ארגומנטים, כאשר הראשון מתייחס למופע של המחלקה, ואילו השני מתייחס האם להציג את הטלפון של הלקוח או לא.

שימו לב !!!
כאשר יש פרמטרים אחד , אין צורך בסוגריים עגולות
כאשר יש שני פרמטרים, חובה לשים את שני הפרמטרים בסוגריים עגולות

Supplier

הטיפוס האחרון הינו supplier, ותפקידו רק להחזיר מידע ללא קבלת מידע.

לדוגמא :
על מנת לקבל מידע על כתובת חיבור לבסיס נתונים נכתוב את השיטה הבאה

```
String getDBConnectionUrl() {
    return "jdbc://localhost:5432/users";
}
```

והשימוש בה

```
System.out.println(getDBConnectionUrl());
```

כאשר נרצה לעבוד עם כתיבת פונצקיות נצהיר על supplier

```
static Supplier<List<String>> getDBConnectionUrlsSupplier = ()-> List.of(
        "jdbc://localhost:5432/users",
        "jdbc://localhost:5432/customer");
```

כלומר הגדרנו שאנו מחזירים בעזרת supplier רשימה מסוג מחרוזת
השתמשנו בביטוי <-() על מנת לציין שאנו נשתמש בביטוי למבדא אבל ללא ארגומנטים, וייצרנו את הרשימה שלנו

שימוש בטיפוס

```
System.out.println(getDBConnectionUrlsSupplier.get());
```

כלומר, הפעם נשתמש בשיטה שנקראת get

לסיכום	
טיפוס	שיטה
Function	apply()
Predicate	test()
Consumer	accept()
Supplier	get()

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

Stream

בעזרת מה שלמדנו עד עכשיו, נוכל לבנות stream, כלומר להזין מידע , לבצע חיתוך בעזרת Predicat, לספור רשומות, לקבל ערכים ייחודיים, לבצע מיון, איסוף והחזרה של המידע לתוך כל אוסף שנרצה, בצורה קלה, פשוטה ומהירה.

על מנת לגשת לקוד, ולקבל את כל המידע כולל הקבצים עם נתוני דמה, ניתן לגשת לכתובת הגיט:
https://github.com/zeev-mindali/functionalProgramming

אז מה זה למעשה streams?
ראינו שיש שני גישות לקבלת נתונים:

- Imperative Approach (גישה הכרחית)
- Declarative Approach (גישה הצהרתית)

בעזרת גישה הכרחית, נבקש לבצע את הפעולות הבאות:

1. קבלת אנשים מתחת לגיל 18
2. נרצה לקבל את עשרת הראשונים

קטע הקוד יראה כך:

```
List<Person> people = MockData.getPeople();
List<Person> youngPeople = new ArrayList<>();
int limit = 10;
int counter = 0;
for (Person person : people) {
    if (person.getAge() <= 18) {
        youngPeople.add(person);
        counter++;
        if (counter == limit) {
            break;
        }
    }
}
youngPeople.forEach(System.out::println);
```

אנו יכולים לראות שהגבלנו ל 10 אנשים ויצרנו אוסף חדש של אנשים צעירים, ולאחר מכן אנו מבצעים איטירציה על האוסף לקבל אנשים מתחת לגיל 18, עד שנגיע לעשרת האנשים הראשונים.
ברגע שהגענו ל10 אנשים, נבצע break.
בסיום הפעולה נדפיס את האוסף החדש שייצרנו.
התוכנה מעט מסורבלת, קשה להבנה מהירה, ואיטית.

לעומת זאת, כאשר נעבוד עם stream, התוכנה הופכת להיות קצרה יותר ומובנת יותר

```
List<Person> people = MockData.getPeople();
List<Person> youngPeople = people.stream()
        .filter(p -> p.getAge() <= 18)
        .limit(10)
        .collect(Collectors.toList());
youngPeople.forEach(System.out::println);
```

בשורה ראשונה יצרנו אוסף של נתוני דמה.
בשורה שנייה יצרנו אוסף של אנשים צעירים, כאשר הכנסנו ישירות את המידע אל האוסף בתנאים שאנו מעוניינים בהם, נפרק את הפקודה לשורות, ונסביר כל שורה:

```
people.stream()
```

יוצר זרם של נתונים מתוך נתוני הדמה

```
.filter(p -> p.getAge() <= 18)
```

קבענו שעבור כל אובייקט שמגיע בזרם, אנו מחפשים גיל נמוך מגיל 18, שימו לב שהשתמשנו פה למעשה בטיפוס predicate, אשר יחזיר true במידה והתנאי מתקיים, במידה והתנאי מתקיים, המערכת תעבור לשלב הבא, במידה ולא היא תעבור לאיטרציה הבאה

```
.limit(10)
```

הגבלנו את הכמות של המידע ל10 אובייקטים בלבד, באובייקט ה-10, המערכת תאסוף את המידע ותסיים את הפעולה.

```
.collect(Collectors.toList());
```

ביקשנו שאת התוצאות המערכת תאסוף בשבילנו לאוסף מסוג List.

```
youngPeople.forEach(System.out::println);
```

לבסוף ביקשנו להדפיס את התוצאה.

כעת הקוד יותר פשוט, מובן, קריא וברור לכל משתמש.
כל שינוי בקוד יהיה פשוט, ובמידה ונרצה לשנות את הכלל לחיתוך עפ"י חתך אחר או כמות משתמשים אחרת, נוכל לעשות זאת בקלות וביעילות.

שימוש בחתכים (Filters)

כאשר נרצה לקבל חיתוך נתונים באוסף, לדוגמא, קבלת כל המכוניות בצבע צהוב שעולות פחות מ20,000 ש״ח, נוכל לבצע חיתוך על האוסף.

```
List<Car> cars = MockData.getCars();

Predicate<Car> carPredicate = car -> car.getPrice() < 20_000.00;
Predicate<Car> yellow = car -> car.getColor().equals("Yellow");

List<Car> carsLessThan20k = cars.stream()
        .filter(carPredicate)
        .filter(yellow)
        .collect(Collectors.toList());

carsLessThan20k.forEach(System.out::println);
```

רשמנו במקרה הזה שני טיפוסים מסוג predicate, כאשר הראשון מגביל את הסכום לפחות מ20,000 ש״ח ואילו החתך השני מחפש צבע ספציפי, צהוב במקרה שלנו.

לבסוף ביקשנו לבצע stream המכיל את שני החתכים שלנו, ואיסוף לתוך אוסף מסוג list.

יכלנו גם לרשום ישירות את טיפוס ה predicate לתוך הstream, ולקצר את הקוד.
שימו לב, שאם היינו צריכים לעבוד בשיטה הישנה, היינו רושמים הרבה יותר שורות קוד.

drop while

אפשרות נוספת היא חיתוך לקבל תוצאה מסויימת כמו הדפסת המספרים הזוגיים באוסף.

```
System.out.println("using filter");
Stream.of(2, 4, 6, 8, 9, 10, 13).filter(n -> n % 2 == 0)
        .forEach(n -> System.out.print(n + " "));
```

והתוצאה :

```
using filter
2 4 6 8 10 12
```

במקרה הזה נקבל את כל המספריים הזוגיים באוסף.
אבל מה אם היינו רוצים לקבל את כל המספרים מהרגע שהתנאי נפל והלאה?
לצורך כך אנו נשתמש ב drop while

```
System.out.println("using dropWhile");
Stream.of(2, 4, 6, 8, 9, 10, 13).dropWhile(n -> n % 2 == 0)
        .forEach(n -> System.out.print(n + " "));
```

והתוצאה:

```
using dropWhile
9 10 13
```

take while

במידה ונרצה לקבל את המספר עד לנקודה הנפילה (מקרה הפוך), נשתמש בפקודה take while

```
System.out.println("using take while");
Stream.of(2, 4, 6, 8, 9, 10, 13).takeWhile(n -> n % 2 == 0)
        .forEach(n -> System.out.print(n + " "));
```

והתוצאה

```
using take while
2 4 6 8
```

find first

ישנם מקרים נרצה לדעת אם נתון מסויים קיים ולהדפיס אותו, או לחילופין להדפיס 1- במידה ולא מצאנו אותו.

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result = Arrays.stream(numbers).filter(n -> n == 7)
        .findFirst()
        .orElse(-1);
System.out.println(result);
```

אנו נקבל את הספרה : 7

במידה ונחפש מספר שלא נמצא באוסף

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int result = Arrays.stream(numbers).filter(n -> n == 70)
        .findFirst()
        .orElse(-1);
System.out.println(result);
```

הפעם המחשב ידפיס : 1-

find any

אם נרצה למצוא כל נתון שיש, לא משנה לכמות הפעמים שהוא מופיע ובמידה ולא מופיע להדפיס 1-

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 10};
int result = Arrays.stream(numbers).filter(n -> n == 9)
        .findAny()
        .orElse(-1);
System.out.println(result);
```

במקרה הזה נקבל 9, מכיוון שהנתון קיים.
בהמשך נלמד גם לספור כמה פעמים הסיפרה 9 מופיעה במערך.

all match

דוגמא טובה נוספת, אנו רוצים לבדוק האם כל האוסף מתחלק ב2 ללא שארית (הכל זוגי), אין יותר פשוט מזה בעזרת streams.

```
int[] even = {2, 4, 6, 8, 10};
boolean allMatch = Arrays.stream(even).allMatch(n -> n % 2 == 0);
System.out.println(allMatch);
```

התשובה כמובן תהיה : true

any match

לעומת זאת, נרצה לדעת האם יש לפחות איבר אחד שהוא לא זוגי במערך

```
int[] evenAndOneOdd = {2, 4, 6, 8, 10, 11};
boolean anyMatch = Arrays.stream(evenAndOneOdd).anyMatch(n -> !(n % 2 == 0));
System.out.println(anyMatch);
```

גם כן התשובה תהיה : true

מכיוון שמצאנו 11, שהוא לא זוגי.

distinct

במידה ונרצה לצור אוסף חדש שיכיל אך ורק ערכים יחודיים, עומדות לרשותנו שתי אופציות.

אפשרות ראשונה שימוש באוסף מסוג set

```
List<Integer> numbers = List.of(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8,
8, 9, 9, 9, 9, 9);
Set<Integer> distinct = numbers.stream().collect(Collectors.toSet());
System.out.println(distinct);
```

התוצאה תהיה כמובן:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

אבל מה אם נרצה אוסף ייחודי לתוך אוסף מסוג אחר, list לדוגמא, אין בעייה

```
List<Integer> numbers = List.of(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8,
8, 9, 9, 9, 9, 9);
List<Integer> distinct =
numbers.stream().distinct().collect(Collectors.toList());
System.out.println(distinct);
```

וגם כאן נקבל את התוצאה הרצוייה

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

grouping – איחוד נתונים לפי נתון מסויים

היינו רוצים לבצע איחוד נתונים לפי נושא מסויים, לדוגמא, להדפיס את כל המכוניות לפי סוג היצרן.
בכתיבה רגילה, היינו צריכים לעבור על כל האוסף , לאסוף את כל היצרנים.
כאשר עובדים עם streams העבודה נעשית יותר פשוטה וקלילה

```
Map<String, List<Car>> map = MockData.getCars()
        .stream()
        .collect(Collectors.groupingBy(Car::getMake));
map.forEach((carType, cars) -> {
    System.out.println("---------------------");
    System.out.println("car type: " + carType);
    System.out.println("---------------------");
    cars.forEach(System.out::println);
});
```

בעזרת אוסף מסוג map, נאסוף את כל סוגי המכוניות

```
Map<String, List<Car>> map = MockData.getCars()
        .stream()
        .collect(Collectors.groupingBy(Car::getMake));
```

לאחר מכן, נדפיס את התוצאות

```
map.forEach((carType, cars) -> {
    System.out.println("---------------------");
    System.out.println("car type: " + carType);
    System.out.println("---------------------");
    cars.forEach(System.out::println);
});
```

מה קורה כאשר אנו רוצים לספור כמה פעמים מופיעה מילה מסוימת או איבר מסוים?
אין בעיה, streams יודע לספור את כמות המופעים:

```
List<String> names = List.of(
        "zeev", "amital", "tal", "tal", "zeev", "tal",
        "jacob", "gabi", "gabi"
);

Map<String, Long> map = names.stream()
        .collect(Collectors.groupingBy(
                Function.identity(),
                Collectors.counting())
        );

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

והתוצאה:

```
{gabi=2, zeev=2, amital=1, jacob=1, tal=3}
```

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

int stream

אחד הדברים שעוזרים לנו כמפתחים הוא לצור אינדקסים של מספרים, כלומר טווח מספרים מסויים
עד היום השתמשנו בלולאת for על מנת לצור טווח שכזה, כפי שמובא בדוגמא הבאה:

```
System.out.println("with fori");
for (int index = 0; index <= 10; index++) {
    System.out.println(index);
}
```

והתוצאה:

```
0 1 2 3 4 5 6 7 8 9 10
```

כעת אנו יכולים להשתמש ביכולת (שכבר קיימת בשפות אחרות כמו פייטון) על מנת לצור מערך מספרים
בצורה מהירה, פשוטה וקלה כאשר הגבול העליון (10) איננו נחשב:

```
System.out.println("with int stream exclusive");
IntStream.range(0, 10).forEach(System.out::println);
```

והתוצאה:

```
0 1 2 3 4 5 6 7 8 9
```

במידה וכן נרצה להשתמש בגבול העליון, פשוט נבקש טווח סגור

```
System.out.println("with int stream inclusive");
IntStream.rangeClosed(0, 10).forEach(System.out::print);
```

והתוצאה:

```
0 1 2 3 4 5 6 7 8 9 10
```

אומנם שימוש במספרים שלמים תמיד נחמד, אבל מה קורה כאשר אנו רוצים לבצע חיתוך לפי אינדקס,
כלומר להציג טווח אינדקסים מסויים, אין יותר קל מזה:

```
List<Person> people = MockData.getPeople();
IntStream.range(0, 5)
        .forEach(index -> {
            System.out.println(people.get(index));
        });
```

כעת המערכת תדפיס לנו את חמשת האנשים הראשונים
(שימו לב, במידה והיינו רוצים גם את אינדקס 5, כלומר 6 מופעים, היינו משתמשים ב (range closed)

```
Person{id=1, firstName='Dixie', lastName='O'Finan', email='dofinan0@huffingtonpost.com',
gender='Female', age=91}
Person{id=2, firstName='Harmon', lastName='Marling', email='hmarling1@moonfruit.com', gender='Male',
age=38}
Person{id=3, firstName='Dallas', lastName='Beynon', email='dbeynon2@rambler.ru', gender='Male', age=61}
Person{id=4, firstName='Carlyle', lastName='Lachaize', email='clachaize3@youtu.be', gender='Male',
age=3}
Person{id=5, firstName='Eula', lastName='Pimm', email='epimm4@google.com.hk', gender='Female', age=54}
```

ובמידה והייתי רוצה לבצע איטרציה על טווח מסויים:

```
IntStream.iterate(0, value -> value + 1)
        .limit(11)
        .forEach(System.out::println);
```

והתוצאה:

```
0 1 2 3 4 5 6 7 8 9 10
```

String Join

כאשר דיברנו על String, הבנו כי מחרוזת היא למעשה מחלקה,
אבל הבעייה ב String היא שהמופעים הם למעשה immutable, כלומר אינם נמחקים מזיכרון ה-heap.
וכאשר אנו מחברים מחרוזות אנו למעשה יוצרים מופעים חדשים בזיכרון ה-heap שאינם נמחקים.

הדרך הנכונה על מנת לאחד מחרוזות היא שימוש ב String Builder
בדוגמא אנו משתמשים באוסף מסוג list על מנת לצור מופע של מחרוזת המכילה את הערכים ורושמת את האות הראשונה בשם כאות גדולה

```
List<String> names = List.of("zeev", "amital", "tal", "gabi", "jacob");

StringBuilder join = new StringBuilder();

for (String name : names) {
    join.append(name.substring(0, 1).toUpperCase())
            .append(name.substring(1))
            .append(", ");
}
System.out.println(join.substring(0, join.length() - 2));
```

והתוצאה:

```
Zeev, Amital, Tal, Gabi, Jacob
```

בעזרת streams, אנו יכולים לבצע את אותה פעולה, אבל בצורה הרבה יותר פשוטה

```
List<String> names = List.of("zeev", "amital", "tal", "gabi", "jacob");
String join = names.stream()
        .map(name -> name.substring(0, 1).toUpperCase() + name.substring(1))
        .collect(Collectors.joining(", "));
System.out.println(join);
```

והתוצאה זהה:

```
Zeev, Amital, Tal, Gabi, Jacob
```

מיונים (sorting)

ראינו כאשר אנו רוצים למיין מופעים באוסף מסוים עומדים לרשותנו שני כלים:

- Comparable
- Comparator

היינו צריכים לממש אחד מכלי המיון על מנת לקבל את התוצאה הרצויה.
וזה בדיוק ההבדל בין Imperative approach לבין Declarative approach.

כעת, אין לנו צרוך להצהיר על שיטות מיון, אלא פשוט להשתמש בהם.

כדי לקבל אוסף ממויין, אנו פשוט נבקש זאת עם השיטה sorted:

```
List<Person> people = MockData.getPeople();
List<String> sorted = people.stream()
        .map(Person::getFirstName)
        .sorted()
        .collect(Collectors.toList());
sorted.forEach(System.out::println);
```

כעת קיבלנו למעשה רשימה של שמות פרטיים ממויינת לפי סדר ה abc.

ואם נרצה להפוך את הסדר? מהאות z לאות a ? פשוט, נבקש מהמחשב להפוך את הסדר בעזרת הפקודה reverseOrder.

```
List<Person> people = MockData.getPeople();
List<String> sorted = people.stream()
        .map(Person::getFirstName)
        .sorted(Comparator.reverseOrder())
        .collect(Collectors.toList());
sorted.forEach(System.out::println);
```

במידה ונרצה למיין לפי יותר ממאפיין אחד, אנו נצור מופע מסוג comparator, ולאחר מכן, נבקש למיין לפי ההגדרות במופע, במקרה שלנו לפי מייל, ולאחר מכן לפי שם פרטי

```
List<Person> people = MockData.getPeople();
//createing a comparator instance
Comparator<Person> comparing = Comparator
        .comparing(Person::getEmail)
        .reversed()
        .thenComparing(Person::getFirstName);
//create list with stream by comparator
List<Person> sort = people.stream()
        .sorted(comparing)
        .collect(Collectors.toList());
sort.forEach(System.out::println);
```

דרך נוספת למיון, היא שימוש ישיר ב comparator, לצורך הדוגמא, נבקש מהמחשב להציג לנו את עשרת המכוניות הכחולות היקרות ביותר מהמכונית היקרה ביותר למכונית הזולה ביותר.

```
List<Car> cars = MockData.getCars();
List<Car> topTen = cars.stream()
        .filter(car -> car.getColor().equalsIgnoreCase("blue"))
        .sorted(Comparator.comparing(Car::getPrice).reversed())
        .limit(10)
        .collect(Collectors.toList());
topTen.forEach(System.out::println);
```

והתוצאה:

```
Car{id=48, make='Hyundai', model='Tiburon', color='Blue', year=2009, price=98896.61}
Car{id=929, make='Isuzu', model='Oasis', color='Blue', year=1998, price=98709.39}
Car{id=571, make='Pontiac', model='Grand Prix', color='Blue', year=1966, price=97355.32}
Car{id=905, make='Land Rover', model='LR2', color='Blue', year=2011, price=96183.04}
Car{id=236, make='GMC', model='Sonoma', color='Blue', year=1994, price=95535.7}
Car{id=561, make='Lexus', model='LS', color='Blue', year=2002, price=94837.79}
Car{id=77, make='Buick', model='LaCrosse', color='Blue', year=2010, price=93295.81}
Car{id=300, make='Audi', model='V8', color='Blue', year=1994, price=93170.86}
Car{id=915, make='Dodge', model='Ram 3500 Club', color='Blue', year=1997, price=93069.74}
Car{id=73, make='Jeep', model='Grand Cherokee', color='Blue', year=2012, price=90199.32}
```

אין יותר פשוט מזה, חתך עפ״י צבע, מיון לפי מחיר הרכב בסדר יורד (מהיקר לזול) ולבסוף הגבלה ל-10 מופעים בלבד.

קבל הערך הגבוה ביותר והנמוך ביותר

על מנת לקבל את הערך הנמוך ביותר באוסף, נכתוב את הפקודה הבאה:

```
List<Integer> numbers = List.of(1, 2, 3, 100, 23, 93, 99);
Integer min = numbers.stream().min(Comparator.naturalOrder()).get();
System.out.println(min);
```

והתוצאה:

```
1
```

ואילו היינו רוצים את הערך הגבוה ביותר באוסף

```
List<Integer> numbers = List.of(1, 2, 3, 100, 23, 93, 99);
Integer max = numbers.stream().max(Comparator.naturalOrder()).get();
System.out.println(max);
```

והתוצאה:

```
100
```

ספירה של מופעים עפ״י חתך מסויים

כאשר אנו רוצים לקבל ספירה של מופעים עפ״י חתך מסויים או יותר, אנו יכולים לבצע חתך מידע, ולאחר מכן לספור את כמות המופעים העונים לחתך

נניח שאנו רוצים לקבל מאוסף מסויים את כמות הרכבים מדגם פורד, ששנת הייצור שלהם מעל 2010.

```
List<Car> cars = MockData.getCars();
long count = cars.stream()
        .filter(car -> car.getMake().equalsIgnoreCase("Ford"))
        .filter(car -> car.getYear() > 2010)
        .count();
System.out.println(count);
```

אנו כותבים בדיוק כמו שאנו רוצים, חתך לקבלת Ford, חתך לשנה מסוימת, וספירה של כל מופע העונה להגדרה של החתכים.

והתוצאה

```
10
```

סטטיסטיקות של אוספים

קבלת ערך נמוך במופע

נרצה לעבור על רשימה של רכבים, ולקבל את הרכב עם המחיר הנמוך ביותר, או במקרה ולא נמצא רכב שעונה לדרישות, נרצה לרשום 0

```
List<Car> cars = MockData.getCars();
double min = cars.stream()
        .mapToDouble(Car::getPrice)
        .min()
        .orElse(0);
System.out.println(min);
```

והתוצאה:

```
5005.16
```

ובמקביל, הרכב הכי יקר ברשימה

```
List<Car> cars = MockData.getCars();
double max = cars.stream()
        .mapToDouble(Car::getPrice)
        .max()
        .orElse(0);
System.out.println(max);
```

והתוצאה

```
99932.82
```

ובמידה ונרצה לקבל את הממוצע המחירים של הרכבים:

```
List<Car> cars = MockData.getCars();
double average = cars.stream()
        .mapToDouble(Car::getPrice)
        .average()
        .orElse(0);
System.out.println(average);
```

והתוצאה

```
52693.19
```

ולבסוף את סכום הכולל של כלל הרכבים

```
List<Car> cars = MockData.getCars();
double sum = cars.stream()
        .mapToDouble(Car::getPrice)
        .sum();
System.out.println(BigDecimal.valueOf(sum));
```

והתוצאה

```
52693199.79
```

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

יצירת אוספים חדשים ע"י מיפוי ערכים

לפעמים נרצה לקחת אוסף מסויים ולהכניס את הערכים לאוסף אחר, המכיל מספר מצומצם יותר של שדות.

```
List<Person> people = MockData.getPeople();
```

לצורך הדוגמא, אנו רוצים לצור מחלקה חדשה שהשדות שלה הם:

```
private final Integer id;
private final String name;
private final Integer age;
```

אנו נקרא למחלקה PersonDTO

DTO = Data To Object

כעת אנו רוצים למפות את המידע המתקבל בjson מנתוני הדמה אל אובייקט מסויים,
לצורך כך נצור פונקציה אשר תדאג לקחת את השדות הרלוונטים

```
Function<Person, PersonDTO> personPersonDTOFunction = person ->
        new PersonDTO(
                person.getId(),
                person.getFirstName(),
                person.getAge());
```

שימו לב שהפונקציה מקבל אובייקט מסוג Person ומחזירה אובייקט מסוג PersonDTO

לבסוף נשתמש בפונקצייה החדשה שייצרנו בתוך streams, כאשר המופעים הם בעלי גיל מעל 20:

```
List<PersonDTO> dtos = people.stream()
        .filter(person -> person.getAge() > 20)
        .map(PersonDTO::map)
        .collect(Collectors.toList());
```

כעת יש לנו אוסף חדש, המכיל 3 שדות בלבד, ובחתך של גיל מעל 20 שנה

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

במידה ונרצה לקבל את הממוצע של מחיר מכונית במגרש מסויים ולהכניס את התוצאה למשתנה מתאים, אין יותר פשוט מזה, פשוט נכתוב את הפקודה הבאה:

```
List<Car> cars = MockData.getCars();
double avg = cars.stream()
        .mapToDouble(Car::getPrice)
        .average()
        .orElse(0);
System.out.println(avg);
```

והתוצאה

```
52693.19
```

לעומת זאת, ייתכן ונקבל אוסף שמורכב מאוספים אחרים, לדוגמא:

```
private static final List<List<String>> arrayListOfNames = List.of(
        List.of("Zeev", "Amital", "Jacob"),
        List.of("Gabi", "Tom", "Eliran"),
        List.of("Omer", "Dan")
);
```

האוסף שייתקבל יהיה

```
[[Zeev, Amital, Jacob], [Gabi, Tom, Eliran], [Omer, Dan]]
```

היינו רוצים לצור אוסף אחד במקום אוסף בתוך אוסף , או במילים אחרות לשטח את האוסף (flat) בצורה הקונבנציונלית היינו מבצעים את הפעולה הבאה:

```
List<String> names = new ArrayList<>();
for (List<String> strings : arrayListOfNames) {
    names.addAll(strings);
}
System.out.println(names);
```

והתוצאה:

```
 [Zeev, Amital, Jacob, Gabi, Tom, Eliran, Omer, Dan]
```

ניתן לבצע זאת בקלות ע״י שימוש בפקודה flatMap

```
List<String> names = arrayListOfNames.stream()
        .flatMap(List::stream)
        .collect(Collectors.toList());
System.out.println(names);
```

ולקבל את אותה תוצאה:

```
[Zeev, Amital, Jacob, Gabi, Tom, Eliran, Omer, Dan]
```

Functional programing

Lambda expression, Functions, Consumers, Predicates, Supplier, Streams

כל הקודים ונתוני הדמה נמצאים בgithub בכתובת:
https://github.com/zeev-mindali/functionalProgramming

זאב מינדלי
מרצה java full stack