

# תוכנה 1 שיעור 12

זרמי מעמקים


ועוד חידושים משומשים

של ג'אווה 8

```
List<String> names = Arrays.asList("yael",  
                                   "dvir",  
                                   "sivan");
```

## מחלקה אנונימית (עם שירות יחיד)

```
Collections.sort(names,  
    new Comparator<String>() {  
        @Override  
        public int compare(String a, String b) {  
            return a.compareTo(b);  
        }  
    }  
);
```



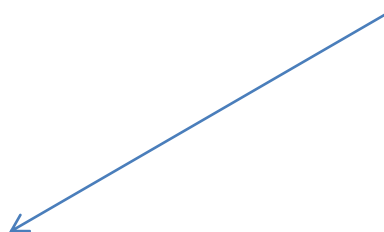
```
System.out.println(names);
```

```
List<String> names = Arrays.asList("yael",  
                                   "dvir",  
                                   "sivan");
```

```
Collections.sort(names,  
    new Comparator<String>() {  
        @Override  
        public int compare(String a, String b) {  
            return a.compareTo(b);  
        }  
    }  
);
```

ביטוי למבדה  
( $\lambda$ )

```
Collections.sort(names,  
    (String a, String b) -> { return a.compareTo(b); }  
);
```



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

```
List<String> names = Arrays.asList("yael",
                                   "dvir",
                                   "sivan");
```

```
Collections.sort(names,
    new Comparator<String>() {
        ...
    }
);
```

```
Collections.sort(names,
    (String a, String b) -> { return a.compareTo(b); }
);
```

```
Collections.sort(names,
    (a, b) -> { return a.compareTo(b); }
);
```

**הסקת טיפוס  
הארגומנטים**

**ותחביר מקוצר לשירותים של שורה אחת**

```
Collections.sort(names, (a, b) -> a.compareTo(b) );
```

# מנשקים פונקציונליים

- מנשקים עם שירות בודד
- שברירי (מה יקרה אם נוסף ל-Comparator עוד שירות?)
- הבטחה לשירות בודד: @FunctionalInterface
- עדיין אפשר להוסיף שירותי default (יכולת ב-8 להוסיף מימוש למנשק!)

```
@FunctionalInterface
public interface SimpleFuncInterface {
    public void doWork();
}
```

```
List<String> names = Arrays.asList("yael",  
                                   "dvir", "sivan");  
  
names.forEach( s -> System.out.println(s) );  
  
names.stream().forEach( s -> System.out.println(s) );  
  
names.stream()  
    .map( s -> s.toUpperCase() )  
    .forEach( s -> System.out.println(s) );  
  
names.stream()  
    .map( String::toUpperCase )  
    .forEach( s -> System.out.println(s) );  
  
names.stream()  
    .map( s-> (int) s.charAt(0) )  
    .forEach( s -> System.out.println(s) );
```

```
List<String> names = Arrays.asList("yael",
                                   "dvir", "sivan");
```

```
names.forEach( s -> System.out.println(s) );
```

```
names.stream().forEach( s -> System.out.println(s) );
```

```
names.stream()
    .map( s -> s.toUpperCase() )
    .forEach( s -> System.out.println(s) );
```

**stream מחזיר**

```
names.stream()
    .map( String::toUpperCase )
    .forEach( s -> System.out.println(s) );
```

**שירות כארגומנט**

```
names.stream()
    .map( s -> (int) s.charAt(0) )
    .forEach( s -> System.out.println(s) );
```

**המרה של טיפוס איברי ה-stream**

# זרמי נתונים (streams)

- מייצגים אוסף או סדרה מופשטת של עצמים
- להבדיל מאוספים (collections) שמייצגים אוסף קונקרטי של עצמים בזיכרון
- יכולים להיות סדורים (sequential) או לא (parallel)
- עצלנים: קונקרטיזציה רק כאשר צריך איבר



```
List<String> names = Arrays.asList("yael",  
                                   "dvir", "sivan");
```

```
names.stream()  
    .filter( s -> s.contains("a"))  
    .forEach( s -> System.out.println(s) );
```

```
names.stream()  
    .filter( s -> s.contains("a"))  
    .sorted()  
    .forEach( s -> System.out.println(s) );
```

**sorted** היא פעולה לא שגרתית על **stream** כי היא צריכה לאסוף את כל איבריו.

# יצירת זרם (אינסופי!)

```
public class NaturalNumbers
    implements Supplier<Integer> {
    private int i = 0;
    public Integer get() { return ++i; }
}

public static void main(String[] args) {
    Stream<Integer> s =
        Stream.generate(new NaturalNumbers());

    s.limit(5)
      .map( x -> x*x )
      .forEach( System.out::println );
}
```

# שימוש נוסף בכל איבר

```
public class NaturalNumbers
    implements Supplier<Integer> {
    private int i = 0;
    public Integer get() { return ++i; }
}

public static void main(String[] args) {
    Stream<Integer> s =
        Stream.generate(new NaturalNumbers());

    s.limit(5)
      .peek( System.out::println )
      .forEach( System.out::println );
}
```

מה יודפס?

# עצלנות סופנית

- זרמים הם עצלנים (lazy)
- שירותים כמו map, peek, וכו' מופעלים על זרם ומחזירים זרם אחר
- כל זמן שלא צריך איברים של הזרם, לא קורה בפועל כלום.
- שירותים סופניים (terminal, כמו forEach, אבל יש עוד) צורכים איברים מהזרם שעליו הם מופעלים וזה גורם לו לייצר את האיברים, לעיתים קרובות על ידי צריכה של איברים מזרם קודם

# תרגיל

```
public class NaturalNumbers
    implements Supplier<Integer> {
    private int i = 0;
    public Integer get() { return ++i; }
}

public static void main(String[] args) {
    Stream<Integer> s =
        Stream.generate(new NaturalNumbers());

    s.limit(5)
      .peek( System.out::println )
      ;
}
```

מה יודפס עכשיו (בלי forEach)?

# אין משמעות בלי סופניות

```
public class NaturalNumbers
    implements Supplier<Integer> {
    private int i = 0;
    public Integer get() { return ++i; }
}

public static void main(String[] args) {
    Stream<Integer> s =
        Stream.generate(new NaturalNumbers());

    s.limit(5)
      .peek( System.out::println )
      ;
}
```

**כלום! האיברים לא נצרכים ולכן לא מיוצרים בכלל**

# תמצות מידע

```
public class NaturalNumbers
    implements Supplier<Integer> {
    private int i = 0;
    public Integer get() { return ++i; }
}
```

```
public static void main(String[] args) {
    Stream<Integer> s =
        Stream.generate(new NaturalNumbers());
```

מחזיר `Optional<T>`

```
s.limit(5)
    .reduce( (x,y) -> x+y )
    .ifPresent( System.out::println );
}
```

# מזרם לקבוצה: איסוף

```
List<Integer> l = s.limit(5)  
                .collect(Collectors.toList());
```



# תכונות של פעולות על זרמים

- כאשר מפעילים פוקציות על זרמים בעזרת `map`, `peek`, וכדומה, צריכות להיות להן שתי תכונות
- אסור להן להשפיע על שייכות לזרם (`non-interfering`); למשל, אם הזרם מייצג איברי קבוצה, אסור לפונקציות הללו להוסיף או לגרוע מהקבוצה
- והן צריכות להיות חסרות מצב (`stateless`) כלומר התוצאה שלהן (ותופעות הלואי, `side effects`, אם יש) צריכות להיות פונקציה של האיבר של הזרם בלבד ולא של מצב של מבנה נתונים אחר בזיכרון

# אופציות

- הטיפוס הגנרי `Optional<T>` מייצג התייחסות בצורה נקייה שמאפשרת להבחין בין `null` ובין התייחסות לעצם
- מה היה קורה אם היינו מייצרים רק אפס מספרים?

```
s.limit(0)
  .reduce( (x,y) -> x+y )
  .ifPresent( System.out::println );
```

## זרמים פרימיטיביים

- כמו בהרבה מקרים אחרים, השוני בין טיפוסים התייחסות לטיפוסים פרימיטיביים (int וכו') מסבך את העניינים
- יש גם `IntStream`, `DoubleStream` שמתנהגים בדומה ל-`Stream<T>` אבל הם זרים
- יש שירותי תמיכה כמו `mapToInt`
- ויש לזרמים הללו שירותים מיוחדים כמו `sum` (מקרה פרטי של `reduce` שראינו)

## נושאים מתקדמים

- flatMap: בהנתן זרם של עצמים (למשל רשימות) מאפשר לייצר זרם מכל אחד (למשל מעבר על הרשימות) ולקבץ את הזרמים לזרם יחיד
- parallelStream: כל זרם ב-JAVA הוא סדור (למשל מה שחוזר מ-Collection.Stream()) או לא סדור (Collection.parallelStream()) שאפשר לפעול על האיברים שלו בכל סדר, על מנת לקבל ביצועים משופרים על מחשב עם כמה מעבדים

# בחזרה לעתיד: חידושים משומשים

- הרעיונות של למבדה (שגרה ללא שם), זרמים (כולל lazy evaluation), ופונקציות ללא תוצאות לואי או מצב הם רעיונות מפרדיגמה בשם תכנות פונקציונלי
- הפרדיגמה הזאת עומדת בבסיס שפות תכנות כגון lisp, scheme, ml, ועוד, שלא זכו לפופולריות
- אבל עכשיו הרעיונות שלהם חילחלו בחזרה לשפות מונחות עצמים כמו ה-"חידושים" האלה בג'אווה

# איסוף מתקדם (הכנות)

```
class Person {  
    private String name;  
    private int age;  
  
    Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

דוגמה של Benjamin Winterberg

# איסוף מתקדם

```
List<Person> persons = Arrays.asList(  
    new Person("Max", 18),  
    new Person("Peter", 23),  
    new Person("Pamela", 23),  
    new Person("David", 12));
```

```
Map<Integer, List<Person>> personsByAge = persons  
    .stream()  
    .collect(Collectors.groupingBy(p -> p.age));
```

```
personsByAge.forEach(  
    (age, p) ->  
        System.out.format("age %s: %s\n", age, p)  
);
```

## איסוף מתקדם 2

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));
```

- ויש עוד אספנים מתקדמים (of ,toMap)



# סיכום השיעור

- למבדה
- זרמים (streams)
- באופן כללי, תכנות פונקציונלי בג'אווה