

# תוכנה 1 בשפת Java

## שיעור מספר 10: חידושים ב

# Java 8

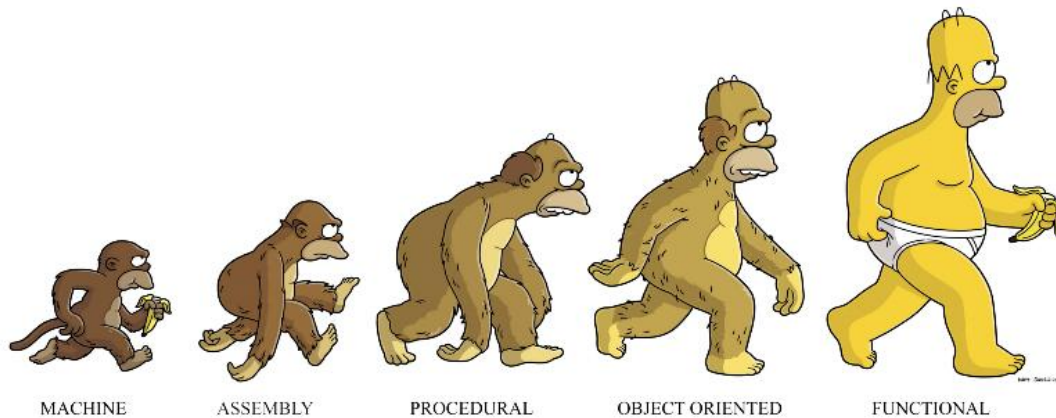


בית הספר למדעי המחשב  
אוניברסיטת תל אביב

# היום בשיעור

■ תכנות פונקציונלי

■ זרמים



from: <https://medium.com/@cscalfani/so-you-want-to-be-a-functional-programmer-part-1-1f15e387e536>

תוכנה 1 בשפת Java  
אוניברסיטת תל אביב

# מנשקים פונקציונליים

■ נרצה למיין רשימת מחרוזות בסדר עולה, על פי אורך המחרוזת.

```
List<String> beatles = Arrays.asList("John",  
                                     "Paul", "George", "Ringo");  
  
Collections.sort(beatles, new Comparator<String>() {  
    @Override  
    public int compare(String a, String b) {  
        return Integer.compare(a.Length(), b.Length());  
    }  
});  
System.out.println(beatles);
```



מחלקה אנונימית עם  
שירות יחיד

# מנשקים פונקציונליים

- מנשק פונקציונלי הוא מנשק בעל מתודה אבסטרקטית אחת בלבד (אין מניעה להוסיף מתודות דיפולטיות/סטטיות)
- החל מ Java 8 ניתן לממש מנשק פונקציונלי באמצעות ביטויי `lambda`.

# מנשקים פונקציונליים

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



```
Comparator<String> cmp = (String x, String y)-> {  
    return Integer.compare(x.Length(), y.Length());  
};  
Collections.sort(beatles, cmp);
```

# מנשקים פונקציונליים

- אפשר להימנע מיצירת המשתנה עבור ה Comparator, ולשלוח את פונקציית ה lambda ישירות ל sort.

```
Collections.sort(beatles,  
    (String x, String y)-> {  
        return Integer.compare(x.Length(), y.Length());  
    });
```



הסקת טיפוסים

```
Collections.sort(beatles,  
    (String x, String y)-> {  
        return Integer.compare(x.Length(), y.Length());  
    });
```

# מנשקים פונקציונליים

- הסקת טיפוסי הארגומנטים נעשית פה בצורה אוטומטית:
- אנחנו ממיינים רשימה של מחרוזות, לכן ה `Comparator` חייב להשוות מחרוזות, והטיפוסים של `x, y` מוסקים בזמן הקומפילציה.

```
Collections.sort(beatles,  
    (x, y)-> {  
        return Integer.compare(x.length(), y.length());  
    });
```



כתיבה מקוצרת עבור  
מימוש בשורה אחת

```
Collections.sort(beatles,  
    (x, y)-> Integer.compare(x.length(), y.length())  
);
```

# מנשקים פונקציונליים

■ אנוטציה (annotation) יעודית המבטיחה שהמנשק מגדיר בדיוק פונקציה אבסטרקטית אחת

```
@FunctionalInterface
public interface I1{
    public void func1(int x);
}
```



# רפרנסים למתודות

## ■ רפרנס למתודה סטטית:

```
List<Integer> ints = Arrays.asList(5,1,2,4,3);  
ints.sort(Integer::compare);
```



```
ints.sort((x,y)->Integer.compare(x, y));
```

## ■ רפרנס למתודת מופע:

```
List<String> strings = Arrays.asList("aa", "Ab", "BA", "Bb");  
strings.sort(String::compareToIgnoreCase);
```



```
strings.sort((x,y)->x.compareToIgnoreCase(y));
```

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html> :דוגמאות נוספות:

# זרמים

- זרם – סדרה מופשטת של אלמנטים התומכים בביצוע פעולות צבירה (aggregation), באופן סדרתי או מקבילי.
- בקורס נדבר על ביצוע פעולות סדרתי.

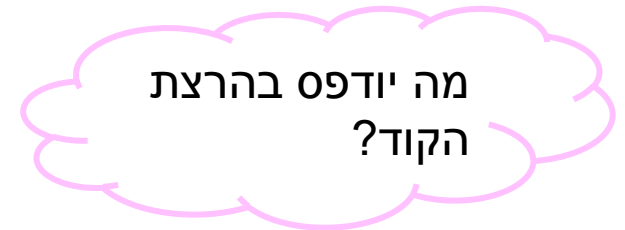
```
List<Integer> ints = Arrays.asList(1,2,3,4,5);  
ints.stream().map(x->x*x)  
        .filter(x->(x>10))  
        .forEach(System.out::println);
```



```
.forEach(x->System.out.println(x));
```

output:

16  
25



# זרמים

- ניתן לחלק את הפעולות על זרמים לשתי קבוצות:
  - פעולות ביניים (intermediate). פעולות אלה מופעלות על זרמים ומחזירות זרם, כך שניתן לשרשר אותן אחת לשניה.
  - פעולות סופניות (terminal). פעולות אלה יופיעו בסוף שרשרת פעולות על זרם, כלומר, לא ניתן לשרשר אחריהן פעולות נוספות על אותו הזרם.

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);  
ints.stream().map(x->x*x)  
           .filter(x->(x>10))  
           .forEach(System.out::println);
```

פעולות ביניים

פעולה סופנית

# עצלנות סופנית

- זרמים הם עצלנים – כל זמן שלא **צורכים** איברים של הזרם, לא קורה כלום.
- מי **צורך** איברים של הזרם? שירותים סופניים. הפעלת שירות סופני בעצם גורמת לזרם לייצר את האיברים שלו.



```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
ints.stream().map(x->{
    System.out.println("mapping " + x);
    return x*x;
})
.filter(x->{
    System.out.println("filtering: " + x);
    return x>10;
})
.forEach(x->{
    System.out.println("terminating: " + x);
});
```

output:

```
mapping 1
filtering: 1
mapping 2
filtering: 4
mapping 3
filtering: 9
mapping 4
filtering: 16
terminating: 16
mapping 5
filtering: 25
terminating: 25
```

מה יודפס?

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
ints.stream().map(x->{
    System.out.println("mapping " + x);
    return x*x;
})
.filter(x->{
    System.out.println("filtering: " + x);
    return x>10;
})
.forEach(x->{
    System.out.println("terminating: " + x);
});
```

מה יודפס אם נוריד את  
הקריאה ל `forEach` ?

# פעולות נוספות על זרמים

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);  
boolean res = ints.stream()  
    .map(x->""+x)  
    .skip(2)  
    .anyMatch(x-> x.equals("1"));
```

שינוי טיפוס האיברים בזרם מ Integer ל String

דילוג על שני האיברים הראשונים בזרם

האם לפחות אחת  
המחרוזות בזרם היא  
המחרוזת "1"?

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);  
boolean res = ints.stream()  
    .limit(1)  
    .allMatch(x-> x ==1);
```

מגבילים את מספר האיברים בזרם ל 1

האם כל האיברים בזרם שווים ל 1?

הפעולות anyMatch, allMatch, noneMatch, forEach בניגוד ל forEach, הן פעולות סופניות, כמו forEach.

```

public static int comparesCounter;
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(5,4,3,2,1,6);
    ints.stream()
        .filter(x->x%2==0)
        .peek(x->{System.out.println("peek " + x);})
        .sorted((x,y)->{
            comparesCounter++;
            System.out.println("comparing: " + x + ", " + y);
            return Integer.compare(x, y);
        })
        .forEach(System.out::println);
    System.out.println("num of compares: " + comparesCounter);
}

```

output:

```

peek 4
peek 2
peek 6
comparing: 2,4
comparing: 6,2
comparing: 6,4
2
4
6
num of compares: 3

```

הפעולה peek מחזירה את הזרם עליו היא מופעלת, ובנוסף, מפעילה על כל איברי הזרם את הפעולה שקיבלה כפרמטר.

הפעולה sorted אינה פעולה שגרתית. על מנת לבצע אותה, יש לאסוף את כל אברי הזרם עליו היא מופעלת!

מה ישתנה אם נבצע את פעולת ה filter אחרי פעולת ה sort?



# תכונות של פעולות על זרמים

- כאשר מפעילים פונקציות על זרמים באמצעות פקודות כמו `map`, `peek` וכדומה, צריכות להיות להן שתי תכונות:
- *non-interfering* – אסור לפעולות לשנות את האוסף עליו מופעל הזרם (הוספת/הורדת אובייקטים).
- *stateless* – חסרות תופעות לוואי. תוצאת הפעולות צריכה להיות תלויה באיבר הזרם עליו היא מופעלת, ולא במצב של שדה/משתנה אחר.

# Optional<T>

- השימוש ב Optional מאפשר לנו לבטא באופן אחיד מצב שבו הפונקציה לא מחזירה ערך.
- ללא שימוש ב Optional היינו צריכים להחזיר ערך ברירת מחדל כלשהו, למשל null או מחרוזת ריקה. הלקוח היה צריך לדעת שיתכן מצב שבו לא יחזור ערך, ולכתוב קוד המטפל בערך ברירת המחדל שנקבע בפונקציה.
- החזרת ערך אופציונלי:

```
public static Optional<String> findStringOfLenK(  
    List<String> strings, int k){  
    for (String s: strings){  
        if (s.length() == k){  
            return Optional.of(s);  
        }  
    }  
    return Optional.empty();  
}
```

# Optional<T>

## שימוש בערך האופציונלי: ■

```
List<String> lst = Arrays.asList("John", "Paul", "George", "Ringo");  
  
Optional<String> strOfLen6 = findStringOfLenK(lst, 6);  
if (strOfLen6.isPresent()){  
    System.out.println(strOfLen6.get());  
}  
  
strOfLen6.ifPresent(System.out::println);  
  
Optional<String> strOfLen3 = findStringOfLenK(lst, 3);  
System.out.println(strOfLen3.orElse("no-value"));
```

output:  
George  
George  
no-value

<https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html> לקריאה נוספת:

# פעולת reduce

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
Optional<Integer> listProduct = ints.stream()
    .reduce((x,y)->x*y);
Optional<Integer> sumOfSquares = ints.stream()
    .map(x->x*x)
    .reduce((x,y)->x+y);
```

- הפעולה reduce היא פעולה סופנית המחזירה ערך אופציונלי.
- במידה והזרם ריק, reduce אינה מחזירה ערך (כלומר, מחזירה ערך אופציונלי ריק)
- אחרת, reduce מחזירה תוצאה של צבירת כל האיברים בזרם באמצעות הפונקציה אותה היא מקבלת כפרמטר.
- הפרמטר ש reduce מקבלת הוא מטיפוס `BinaryOperator<T>`
- קיימת העמסה לפונקציה reduce עם פרמטרים נוספים.

# הצבת טיפוסים גנריים בירושה/מימוש

- המנשק BinaryOperator הוא מנשק פונקציונלי. את הפונקציה האבסטרקטית שלו apply הוא יורש מהמנשק BiFunction:

המנשק BiFunction מגדיר שלושה פרמטרים גנריים (T,U,R), בעוד שהמנשק BinaryOperator מגדיר פרמטר גנרי אחד בלבד. לכן, BinaryOperator נדרש לבצע הצבה לתוך הפרמטרים T,U,R של BiFunction. הוא מציב בשלושתם את ערכו של T, הפרמטר הגנרי שלו. המשמעות: הפונק' apply של BinaryOperator מופעלת על שני איברים מאותו הטיפוס ומחזירה את אותו הטיפוס.

```
@FunctionalInterface  
public interface BinaryOperator<T>  
    extends BiFunction<T,T,T>
```

```
@FunctionalInterface  
public interface BiFunction<T,U,R>
```

Represents a function that accepts two arguments and produces a result.

This is a functional interface whose functional method is `apply(Object, Object)`.

## Method Summary

All Methods

Instance Methods

Abstract Methods

Default Methods

Modifier and Type

Method and Description

R

`apply(T t, U u)`

Applies this function to the given arguments.

# יצירת זרם (אינסופי)

■ המנשק `Supplier<T>` מתאר זרמים של איברים מטיפוס `T`

```
@FunctionalInterface  
public interface Supplier<T>
```

## Method Summary

All Methods

Instance Methods

Abstract Methods

Modifier and Type

Method and Description

T

`get()`  
Gets a result.

■ מימוש לדוגמא של המנשק:

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

# יצירת זרם (אינסופי)

## שימוש בזרם האינסופי

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());  
s.limit(5).reduce((x,y)->Math.max(x,y))  
                .ifPresent(System.out::println);
```

ifPresent מופעל על  
Optional שחוזר מ reduce

output:  
5

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());  
s.map(x->x+1).forEach(System.out::println);
```

מה יקרה בהרצת  
הקוד הבא?

# יצירת זרם (אינסופי)

■ מבין המספרים המתחלקים ב 7, האם קיים מספר המתחלק ב 10?

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());  
System.out.println(s.filter(x -> x % 7 == 0)  
                    .anyMatch(x -> x % 10 == 0));
```

output:  
true

■ מבין המספרים המתחלקים ב 7 והקטנים מ 70, האם קיים מספר המתחלק ב 10?

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());  
System.out.println(s.filter(x -> x % 7 == 0)  
                    .filter(x -> x < 70)  
                    .anyMatch(x -> x % 10 == 0));
```

הקוד הזה נכנס  
ללולאה אינסופית!



# איסוף

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());  
List<Integer> ints = s.limit(5)  
    .map(x->x*x)  
    .collect(Collectors.toList());  
System.out.println(ints);
```

Output:  
[1, 4, 9, 16, 25]

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());  
Double d = s.limit(5).collect(Collectors.averagingDouble(x->x*x));  
System.out.println(d);
```

Output:  
11.0

הפונקציה `averagingDouble` מפעילה את  
הפונקציה שמקבל כפרמטר על אברי הזרם  
ומחשבת את הממוצע שלהם

# איסוף מתקדם

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());  
Map<Boolean, List<Integer>> partition = s.limit(5)  
    .map(x->x*x)  
    .collect(Collectors.partitioningBy(x->x>10));  
System.out.println(partition);
```

output:

```
{false=[1, 4, 9], true=[16, 25]}
```

```
List<String> beatles = Arrays.asList("John", "Paul", "George", "Ringo");  
Map<Integer, List<String>> groups = beatles.stream()  
    .collect(Collectors.groupingBy(x->x.length()));  
System.out.println(groups);
```

output:

```
{4=[John, Paul], 5=[Ringo], 6=[George]}
```