

תוכנה 1 בשפת Java

שיעור מספר 10: תכנות

פונקציונלי וזרמים



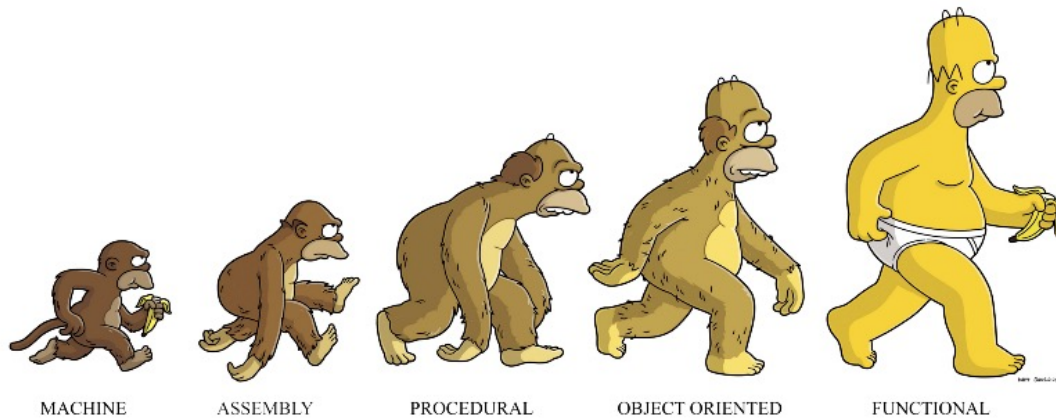
בית הספר למדעי המחשב
אוניברסיטת תל אביב

היום בשיעור

■ תכנות פונקציונלי

■ זרמים

■ enum



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

תוכנה 1 בשפת Java
אוניברסיטת תל אביב

מחלקה אנונימית - תזכורת (?)

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

- מצד אחד – רמאות! אנחנו מפעילים `new` על מנשק.
- מצד שני – מילאנו את כל ה"חובות" כלפי המנשק. בעת יצירת האובייקט אנחנו מממשים את השירות `compare`.
- מדוע אנונימית? כיוון ש `C` הוא מטיפוס דינאמי ששמו לא ידעו.
- זה לא יכול להיות `Comparator`.

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 אבטסטרקטית 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:

דוגמא מהחיים האמיתיים

```
public class Node{  
    public Node left;  
    public Node right;  
    public int value;
```

```
public class Tree{  
    Node root;  
  
    public int sumLeftValues(){ ??? }  
  
    public int sumRightValues(){ ??? }  
}
```

```
public Node(int value, Node left, Node right){  
    this.left = left;  
    this.right = right;  
    this.value = value;  
}
```

```
public Node getLeft(){ return this.left; }
```

```
public Node getRight(){ return this.right;}
```

```
public int getValue(){ return this.value; }
```

```
}
```

פתרון 1

```
public class Tree{
    Node root;
    public int sumLeftValues(){
        int sum = 0;
        Node node = this.root;
        while(node != null){
            sum += node.getValue();
            node = node.getLeft();
        }
        return sum;
    }

    public int sumRightValues(){
        int sum = 0;
        Node node = this.root;
        while(node != null){
            sum += node.getValue();
            node = node.getRight();
        }
        return sum;
    }
}
```



שכפול קוד!!!!

פתרון 2

```
private int sumValues(boolean isLeft){
    int sum = 0;
    Node node = this.root;
    while(node != null){
        sum += node.getValue();
        if (isLeft){
            node = node.getLeft();
        }
        else{
            node = node.getRight();
        }
    }
    return sum;
}
```

מימוש זה טוב יותר מהמימוש המקורי (אין שכפול קוד), אבל זה עדין לא פתרון מספיק טוב.

- מה היינו עושים אם היו 4 צמתים שיוצאים מכל צומת?
- מה אם המימוש של `sumValue` היה מורכב יותר, והיה מצריך פיצול ביותר מפעולה אחת?

```
public int sumLeftValues(){ return sumValues(true); }
```

```
public int sumRightValues(){ return sumValues(false); }
```

פתרון 3

```
private int sumValues(func){
    int sum = 0;
    Node node = this.root;
    while(node != null){
        sum += node.getValue();
        node = func(node)
    }
    return sum;
}
```

היינו רוצים לשלוח ל sumVal פונקציה שמחלצת את הצומת הרלוונטי מה Node. למשל משהו כזה, אם היינו מערבבים גם פייתון:

וב Java נוכל להשתמש במנשק פונקציונלי!

```
public int sumLeftValues(){
    return sumValues(lambda node: node.left());
}

public int sumRightValues(){
    return sumValues(lambda node: node.right());
}
```

דוגמא מהחיים האמיתיים

מה היינו עושים את הקוד המשותף היה מכיל שורות כאלה:

```
public void handleLeft(Node node){
    /* shared code */
    doSomethingWith(node.left());
    /* shared code */
    doSomethingElseWith(node.right());
    /* shared code */
}
```

```
public void handleRight(Node node){
    /* shared code */
    doSomethingWith(node.right());
    /* shared code */
    doSomethingElseWith(node.left());
    /* shared code */
}
```

במקרה הזה אפשר היה להגדיר מנשק (לא פונקציונלי) עם שתי פונקציות: `getFirstChild`, `getSecondChild` ועבור `Right` בצורה הפוכה.

דוגמא מהחיים האמיתיים

אופציה פחות טובה, אבל עדיפה על שכפול קוד:

```
public void handle(Node node, Boolean isLeft){
    Node firstChild, secondChild;
    if (isLeft){
        firstChild = node.left();
        secondChild = node.right();
    }
    else{
        firstChild = node.right();
        secondChild = node.left();
    }
    /* shared code */
    doSomethingWith(firstChild);
    /* shared code */
    doSomethingElseWith(secondChild);
    /* shared code */
}
```

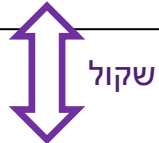
```
public void handleLeft(Node node){
    handle(node, true);
}
```

```
public void handleRight(Node node){
    handle(node, false);
}
```


זרמים

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

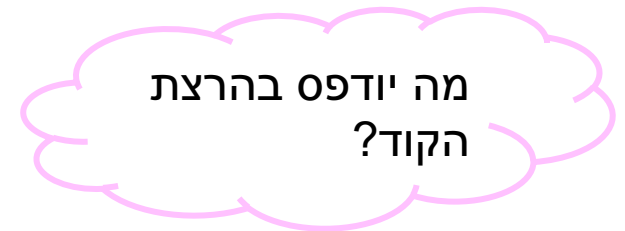
```
List<Integer> ints = Arrays.asList(1,2,3,4,5);  
ints.stream().map(x->x*x)  
    .filter(x->(x%2 == 0))  
    .forEach(System.out::println);
```



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

output:

4
16



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%2 == 0))  
           .forEach(System.out::println);
```

פעולות ביניים

פעולה סופנית

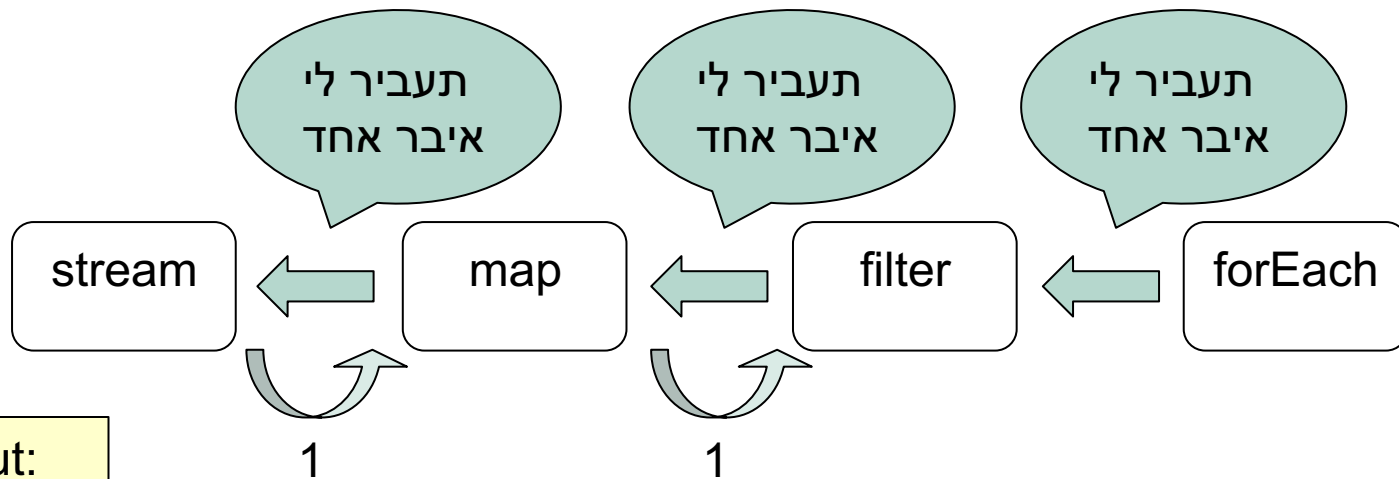
עצלנות סופנית

- 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.



אז איך זה בעצם עובד?

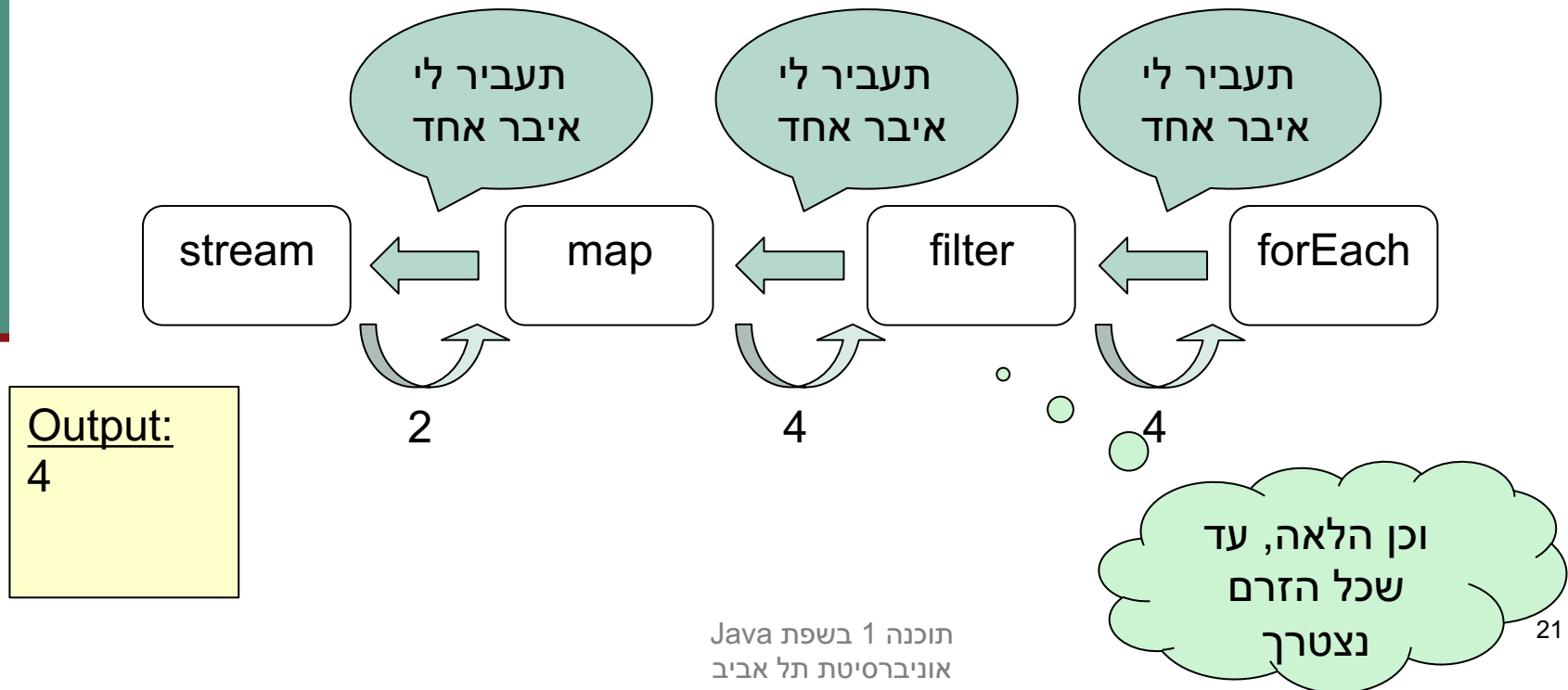
```
List<Integer> ints = Arrays.asList(1,2,3,4,5);  
ints.stream().map(x->x*x)  
          .filter(x->(x%2 == 0))  
          .forEach(System.out::println);
```



Output:

אז איך זה בעצם עובד?

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);  
ints.stream().map(x->x*x)  
        .filter(x->(x%2 == 0))  
        .forEach(System.out::println);
```



```
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> product = 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.
- אם הזרם מכיל איבר יחיד, פעולת ה reduce תחזיר את האיבר הזה.
- 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);
```

output:
5

ifPresent מופעל על
Optional שחוזר מ reduce

```
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));
```

מה יקרה בהרצת
הקוד הבא?

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]}
```

Enumerated Types

טיפוסי מנייה

בית הספר למדעי המחשב
אוניברסיטת תל אביב



Enumerated Types

טיפוסים שכל מופעיהם קבועים וידועים מראש שכיחים מאוד בעולם התוכנה:

```
public class PlayingCard {  
  
    // pseudo enumerated type  
    public static final int SUIT_SPADES = 0;  
    public static final int SUIT_HEARTS = 1;  
    public static final int SUIT_CLUBS = 2;  
    public static final int SUIT_DIAMONDS = 3;  
  
    private int suit;  
    private int rank;  
  
    public PlayingCard(int suit, int rank) {  
        this.suit = suit;  
        this.rank = rank;  
    }  
  
    public String getSuitName() { /* next slide */ }  
}
```

תוכנה 1 בשפת ג'אווה

Enumerated Types

```
public class PlayingCard {  
    // previous code  
    public String getSuitName() {  
        String name = "";  
        switch (suit) {  
            case SUIT_SPADES:  
                name = "Spades";  
                break;  
            case SUIT_HEARTS:  
                name = "Hearts";  
                break;  
            case SUIT_CLUBS:  
                name = "Clubs";  
                break;  
            case SUIT_DIAMONDS:  
                name = "Diamonds";  
                break;  
            default:  
                System.err.println("Invalid suit.");  
        }  
        return name;  
    }  
}
```

הוספת ערך חדש של suit מצריכה שינויים במספר מקומות!
(הוספת קבוע, הוספת case בחדש ב getSuitName).

Enumerated Types

ואולם מימוש טיפוסים אלו בצורה זו אינו בטוח ויש לו חסרונות
נוספים

```
public class TestPlayingCard {  
    public static void main(String[] args) {  
  
        PlayingCard card1 = new PlayingCard(PlayingCard.SUIT_SPADES, 2);  
  
        System.out.println("The rank of card1 is: " + card1.getRank());  
  
    }  
}
```

Enumerated Types

ואולם מימוש טיפוסים אלו בצורה זו אינו בטוח ויש לו חסרונות
נוספים

```
public class TestPlayingCard {  
    public static void main(String[] args) {  
  
        PlayingCard card1 = new PlayingCard(PlayingCard.SUIT_SPADES, 2);  
  
        System.out.println("The rank of card1 is: " + card1.getRank());  
  
        // You can create a playing card with a bogus suit.  
        PlayingCard card2 = new PlayingCard(47, 2);  
  
    }  
}
```

אין שמירה על בטיחות טיפוסים (type safety)!
היינו רוצים לאתחל קלף רק עם אחד מ 4 הערכים האפשריים עבור suit.

Enumerated Types

ואולם מימוש טיפוסים אלו בצורה זו אינו בטוח ויש לו חסרונות
נוספים

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

        PlayingCard card1 = new PlayingCard(PlayingCard.SUIT_SPADES, 2);

        System.out.println("The rank of card1 is: " + card1.getRank());

        // You can create a playing card with a bogus suit.
        PlayingCard card2 = new PlayingCard(47, 2);

        System.out.println("card2 is the " + card2.getRank() + " of "
            + card2.getSuitName());
    }
}
```

אין שמירה על מרחב השמות!
הקשר בין סוג הקלף לייצוג המחרוזתי אינו חלק מהמצב הפנימי של suit
(אין הכמסה – encapsulation)

Enumerated Types

לסיכום: למימוש טיפוסים מניה בצורה זו כמה חסרונות:

- אינו שומר על בטיחות טיפוסים (Not typesafe)
- אינו שומר על מרחב שמות
- הקשר בין סוג הקלף לייצוג המחרוזתי לא חלק מהמצב הפנימי (אין הכמסה – encapsulation)
- הוספת ערך חדש לטיפוס מורכבת
- דורשת שינויים במספר מקומות

New Enumerated Types

- החל ב Java 5.0 התווסף לשפה המבנה **enum**
- הפותר את בעיית בטיחות הטיפוסים

```
public enum Suit {  
    SPADES,  
    HEARTS,  
    CLUBS,  
    DIAMONDS  
}
```

New Enumerated Types

```
public class PlayingCard2 {
```

```
    private Suit suit; ←  
    private int rank;
```

המבנה החדש פותר את בעיית בטיחות הטיפוסים

```
    public PlayingCard2(Suit suit, int rank) {  
        this.suit = suit;  
        this.rank = rank;  
    }
```

```
    public Suit getSuit() {  
        return suit;  
    }
```

```
}
```

New Enumerated Types

```
public class PlayingCard {  
    // previous code  
    public String getSuitName() {  
        String name = "";  
        switch (suit) {  
            case SUIT_SPADES:  
                name = "Spades";  
                break;  
            case SUIT_HEARTS:  
                name = "Hearts";  
                break;  
            case SUIT_CLUBS:  
                name = "Clubs";  
                break;  
            case SUIT_DIAMONDS:  
                name = "Diamonds";  
                break;  
            default:  
                System.err.println("Invalid suit.");  
        }  
        return name;  
    }  
}
```

אפשר להשתמש ב enum
בבלוק switch-case

עדין, בעיית מרחב השמות נשמרת, וגם הוספת סוג קלף חדש מצריכה שינוי במספר מקומות.

New Enumerated Types

```
public class TestPlayingCard2 {  
    public static void main(String[] args) {  
  
        PlayingCard2 card1 = new PlayingCard2(Suit.SPADES, 2);  
  
        System.out.println("card1 is the " + card1.getRank() +  
                            " of " + card1.getSuitName());  
  
        // PlayingCard2 card2 = new PlayingCard2(47, 2);  
        // This will not compile.  
    }  
}
```

איך נפתור את שאר הבעיות?

ב Java כמעט כל דבר הוא עצם - על כן, הרחיבו גם את הקונספט של enum להיות מעין מחלקה (עם שדות, מתודות, בנאים...)

טיפוס מנייה כמחלקה

```
public enum Suit {  
    SPADES("Spades"), ← קריאה לבנאי  
    HEARTS("Hearts"),  
    CLUBS("Clubs"),  
    DIAMONDS("Diamonds");  
  
    private final String name; ← שדה  
  
    private Suit(String name) { ← בנאי  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

כעת אין צורך לשלוף את ייצוג המחלקה כמחרוזת מבחוץ

שימוש בתכונות של טיפוס מניה

```
public class TestPlayingCard3 {  
    public static void main(String[] args) {  
  
        PlayingCard2 card1 = new PlayingCard2(Suit.SPADES, 2);  
        System.out.println("card1 is the " + card1.getRank() +  
            " of " + card1.getSuit().getName());  
  
        // NewPlayingCard2 card2 = new NewPlayingCard2(47, 2);  
        // This will not compile.  
    }  
}
```


התנהגות פולימורפית

```
public enum ArithmeticOperator {
    // The enumerated values
    ADD, SUBTRACT, MULTIPLY, DIVIDE;

    // Value-specific behavior using a switch statement
    public double compute(double x, double y) {
        switch(this) {
            case ADD:         return x + y;
            case SUBTRACT:    return x - y;
            case MULTIPLY:    return x * y;
            case DIVIDE:      return x / y;
            default:          throw new AssertionError(this);
        }
    }
}
```

Real Polymorphism

```
public enum ArithmeticOperator2 {  
    ADD {  
        public double compute(double x, double y) {  
            return x + y;  
        }  
    },  
    SUBTRACT {  
        public double compute(double x, double y) {  
            return x - y;  
        }  
    },  
    MULTIPLY {  
        public double compute(double x, double y) {  
            return x * y;  
        }  
    },  
    DIVIDE {  
        public double compute(double x, double y) {  
            return x / y;  
        }  
    };  
    public abstract double compute(double x, double y);  
}
```

התנהגות פולימורפית

```
public class SomeClient {
    ...

    // Test case for using this enum
    public static void main(String args[]) {

        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);

        for(ArithmeticOperator op : ArithmeticOperator.values())
            System.out.printf("%f %s %f = %f\n",
                               x, op, y, op.compute(x,y));
    }
}
```

Bit Flags

- לעיתים לעצמים יש מס' מאפיינים/תכונות
- לגבי כל עצם יכולות להתקיים כל התכונות, חלקן או אף אחת מהן
- למשל צורה גיאומטרית יכולה להיות
 - קמורה, קעורה, מלאה, חלולה, ישרה, עגולה, צבעונית...
- איך ניתן לייצג זאת ביעילות? בנוחות?

Bit Flags

דרך אחת – נשמור משתנה בוליאני לכל מאפיין

```
boolean isConvex;  
boolean isFull;  
...
```

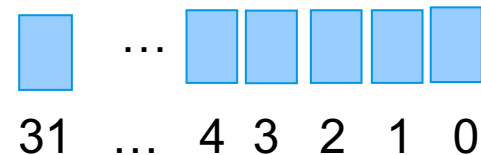
דרך שנייה – נשתמש ב- Bit Flags

```
int shapeAttributes;
```

```
public static final int fullMask = 0x01; // 000...0001  
public static final int convexMask = 0x02; // 000...0010  
public static final int straightMask = 0x04; // 000...0100
```

```
boolean isConvex() {  
    return (shapeAttributes & convexMask) != 0;  
}
```

תוכנה 1 בשפת ג'אווה



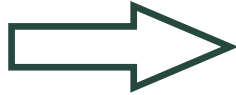
EnumSet

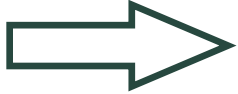
- בג'אווה 5 נוסף מימוש חדש ל-Set המבוסס על Enum
- כל הערכים בסט חייבים לבוא מ-Enum מוגדר כבר, או כזה המוגדר ביצירת הסט
- פנימית, הערכים מוחזקים כביטים, ז"א מאד יעילים

```
enum ShapeAttributes {  
    FULL, CONVEX, STRAIGHT, COLORED  
}
```

למשל לצורה שלנו... ■

```
public class Testing {  
    public static void main(String[] args) {  
        Set<ShapeAttributes> s1 = EnumSet.of(ShapeAttributes.COLORED);  
        if (s1.contains(ShapeAttributes.CONVEX))  
            System.out.println("S1 is convex");  
  
        Set<ShapeAttributes> s2 = EnumSet.of(ShapeAttributes.CONVEX,  
                                             ShapeAttributes.FULL);  
        if (s2.contains(ShapeAttributes.CONVEX))  
            System.out.println("S2 is convex");  
  
        Set<ShapeAttributes> s3 = EnumSet.allOf(ShapeAttributes.class);  
        System.out.println(s3);  
    }  
}
```

 S2 is convex

 [CONVEX, FULL, STRAIGHT, COLORED]

EnumMap

אחיו החורג של EnumSet ■

```
enum Colors {  
    RED, GREEN, BLUE, YELLOW  
}  
  
public class Testing {  
    public static void main(String[] args) {  
        Map<Colors,String> m = new EnumMap<Colors,  
                                           String>(Colors.class);  
  
        m.put(Colors.RED, "Red");  
        m.put(Colors.BLUE, "Blue");  
  
        System.out.println(m);  
    }  
}
```


סיכום

- טיפוס מניה הן סוכר תחבירי למחלקות אשר כל המופעים שלהן ידועים ונוצרו מראש
- בשונה משפות תכנות אחרות, טיפוס המניה הוא עצם לכל דבר, ובפרט יש לו שדות ושרותים
- מקרה פרטי של שימוש בטיפוסי מניה הוא עבור ייצוג תכונות של טיפוס כלשהו כאשר אוסף התכונות ידוע מראש. המחלקות EnumSet ו- EnumMap מייעלות את השימוש ב enum למטרה זו