תוכנה 1 בשפת Java

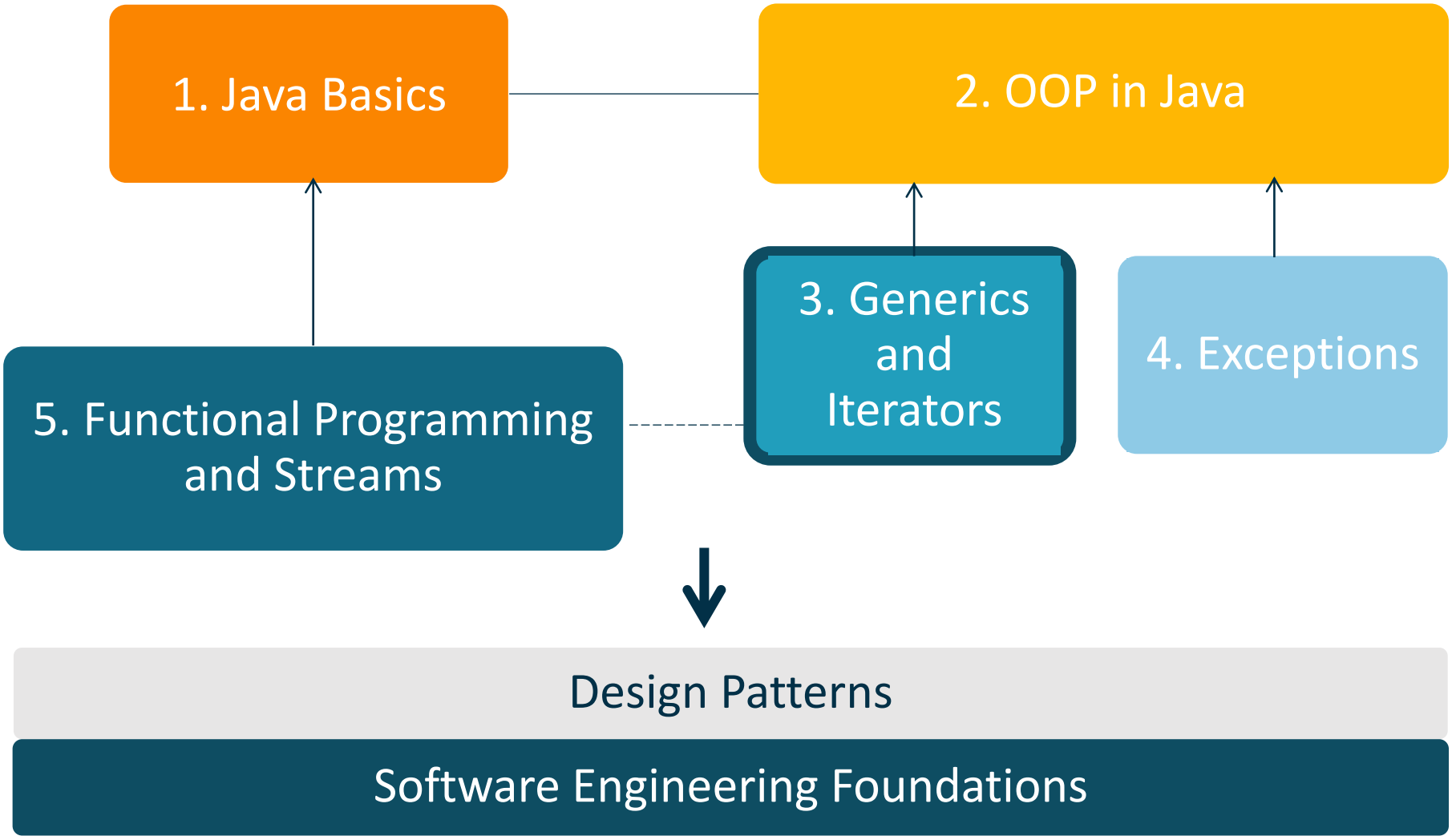
שיעורים מספר 7-8: טיפוסים גנריים ואיטרטורים

מיכל קליינבורט

בית הספר למדעי המחשב

אוניברסיטת תל אביב

נושאי הקורס


1. Java Basics
2. OOP in Java
3. Generics and Iterators
4. Exceptions
5. Functional Programming and Streams
Design Patterns
Software Engineering Foundations

תוכנית לשבוע הקרוב

- מבנים מקושרים
- הכללה של המבנה ע"י הכללת טיפוסים (Generics)
- מחלקות פנימיות (או ייצוג "הכרות אינטימית" בשפת התכנות)
- הפשטת מעבר סידרתי על נתונים (איטרטורים)
- הספקת טיפוסים במשתנים מקומיים
- עוד על generics

התכנית לשבוע הקרוב

- מבנים מקושרים
- הכללה של המבנה ע"י הכללת טיפוסים (Generics)
- מחלקות פנימיות (או ייצוג "הכרות אינטימית" בשפת התכנות)
- הפשטת מעבר סידרתי על נתונים (איטרטורים)
- הספקת טיפוסים במשתנים מקומיים
- עוד על generics

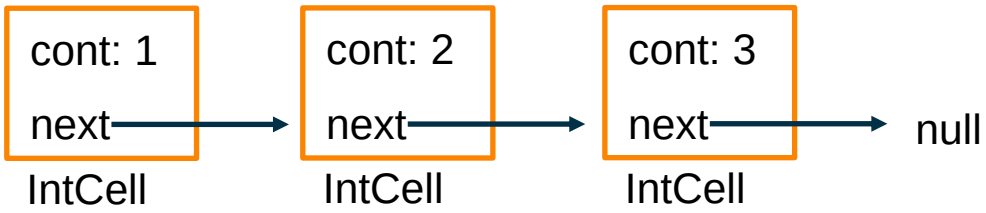
מבנים מקושרים

- כדי לייצג מבנים מקושרים, כגון רשימה מקושרת, עץ, וכדומה, מגדירים מחלקות שכוללות שדות שמתייחסים לעצמים נוספים מאותה מחלקה (ולפעמים גם למחלקות נוספות)

- כדוגמה פשוטה ביותר, נגדיר מחלקה **`IntCell`** שעצמים בה מייצגים אברים ברשימות מקושרות של שלמים
  - המחלקה מייצאת **בנאי** ליצירת עצם כאשר התוכן (שלם) והאבר הבא הם פרמטרים
  - המחלקה מייצאת **שאילתות** עבור התוכן והאבר הבא, **ופקודות** לשינוי האבר הבא, ולהדפסת תוכן הרשימה מהאבר הנוכחי
  - השדות מוגדרים כפרטיים – מוסתרים מהלקוחות

class IntCell

```
public class IntCell {

    private int cont;
    private IntCell next;

    public IntCell(int cont, IntCell next) {
        this.cont = cont;
        this.next = next;
    }

    public int cont() {
        return cont;
    }
```



class IntCell

```
    public IntCell next() {
        return next;
    }

    public void setNext(IntCell next) {
        this.next = next;
    }

    public void printList() {
        System.out.print("List: ");

        for (IntCell y = this; y != null; y = y.next())
            System.out.print(y.cont() + " ");

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

משתנה העזר של הלולאה
הוא מטיפוס IntCell

מחלקה לביצוע בדיקות

- כדי לבדוק שהמחלקה שכתבנו פועלת כנדרש, נכתוב מחלקה התחלתית לבדיקה, שתכיל השרות הראשי **`main`**
- עלינו לבחור מקרי בדיקה שמכסים אפשרויות שונות כדי שנוכל לגלות שגיאות (אם יש)
- חשוב! שגיאות של מחלקה או שרות מוגדרות בהקשר של *החוזה* של המחלקה. אם למחלקה (או לשרות שלה) אין חוזה מפורש לא ברור מהי ההתנהגות ה"נכונה" במקרי קצה
- בהרצאה היום נסתפק באינטואיציה שיש לנו לגבי רשימות מקושרות

  (בדיקות הן נושא נרחב)

מחלקה לביצוע בדיקות

```
public class Test {

    public static void main(String[] args){
        IntCell x = null;
        IntCell y = new IntCell(5,x);
        y.printList();
        IntCell z = new IntCell(3,y);
        z.printList();
        z.setNext(new IntCell(2,y));
        z.printList();
        y.printList();
    }
}
```

מחלקה לביצוע בדיקות – הפלט

```
List: 5
List: 3 5
List: 3 2 5
List: 5
```

- איך ניצור מבנה מקושר של תווים? או של מחרוזות?
- יצירת מחלקה חדשה בגון **`StringCell`** או **`CharCell`** תשכפל הרבה מהלוגיקה הקיימת ב **`IntCell`**
- יש צורך **בהפשטת** הטיפוס **`int`** מטיפוס הנתונים **`Cell`**
- היינו רוצים **להכליל** את הטיפוס **`Cell`** לעבוד עם כל סוגי הטיפוסים

התכנית לשיעורים הקרובים

- מבנים מקושרים
- הכללה של המבנה ע"י הכללת טיפוסים (Generics)
- מחלקות פנימיות (או ייצוג "הכרות אינטימית" בשפת התכנות)
- הפשטת מעבר סידרתי על נתונים (איטרטורים)
- הספקת טיפוסים במשתנים מקומיים
- עוד על generics

מחלקות ושרותים מוכללים (גנריים)

- החל מגירסה 1.5 (נקראת גם 5.0) ג'אווה מאפשרת הגדרת מחלקות גנריות ושרותים גנריים (Generics)

- מחלקה גנרית מגדירה **טיפוס גנרי**, שמציין אחד או יותר **משתני טיפוס** (type variables) בתוך סוגריים משולשים

- עקב ההוספה המאוחרת לשפה (והדרישה שקוד שנכתב קודם יוכל לעבוד ביחד עם קוד חדש), ומשיקולים של יעילות המימוש, כללי השפה לגבי טיפוסים גנריים הם מורכבים

מחלקות ושרותים מוכללים (גנריים)

- רעיון דומה קיים גם בשפת התכנות C++
  - ב C++ נקראת תכונה זו תבנית (template)

- אנחנו נציג רק את המקרה הפשוט

- דוגמה ראשונה – הכללה של המחלקה **`IntCell`** לייצוג תא שתוכנו מטיפוס פרמטרי **`T`**, כך שכל התאים ברשימה הם מאותו הטיפוס

Cell <T>

```
public class Cell <T> {
    private T cont;
    private Cell <T> next;

    public Cell (T cont, Cell <T> next) {
        this.cont = cont;
        this.next = next;
    }
```

הגדרת הפרמטר

המשמעות של שני הביטויים היא שונה !

שימוש בפרמטר הגנרי T

Cell <T>

```
public T cont() {
    return cont;
}

public Cell <T> next() {
    return next;
}

public void setNext(Cell <T> next) {
    this.next = next;
}
```

Cell <T>

```
    public void printList() {
        System.out.print("List: ");
        for (Cell <T> y = this; y != null; y = y.next())
                System.out.print(y.cont() + " ");
        System.out.println();
    }
}
```

מה השתנה במחלקה?

- לכותרת המחלקה נוסף משתנה הטיפוס `T`
  - מקובל ששמות משתני טיפוס הם אות גדולה אחת אולם זו אינה דרישה תחבירית, ניתן לקרוא למשתנה הטיפוס בשם משמעותי

- הטיפוס שמוגדר הוא `Cell <T>`

- הטיפוס של כל שדה, פרמטר, משתנה זמני, וכל טיפוס מוחזר של שרות שהיה `int` יוחלף ב `T`

- הטיפוס של כל שדה, פרמטר, משתנה זמני, וכל טיפוס מוחזר של שרות שהיה `IntCell` יוחלף ב `Cell<T>`

שימוש בטיפוס גנרי

- כדי להשתמש בטיפוס גנרי יש לספק, בהצהרה על משתנה ובקריאה לבנאי(*), טיפוס קונקרטי עבור כל משתנה טיפוס שלו.
  - לדוגמה: `Cell <String>`

- באנלוגיה להגדרת שרות וקריאה לו, משתנה טיפוס בהגדרת המחלקה מהווה מעין פרמטר פורמלי, והטיפוס הקונקרטי הוא מעין פרמטר אקטואלי.

(*) בהמשך הקורס ניראה שאפשר להימנע מהגדרת הפרמטר הגנרי בקריאה לבנאי.

שימוש בטיפוס גנרי

- הטיפוס הקונקרטי חייב להיות **טיפוס הפנייה**, כלומר אינו יכול להיות פרימיטיבי
- אם רוצים ליצור למשל תאים שתוכנם הוא מספר שלם, **לא ניתן** לכתוב `Cell <int>`
- לצורך זה נזדקק לטיפוסים עוטפים (wrapper type)

טיפוסים עוטפים (wrappers)

- לכל טיפוס פרימיטיבי קיים בג'אווה טיפוס הפנייה מתאים:
  - ל- **`float`** העוטף **`Float`** , ל- **`double`** העוטף **`Double`** וכו'
  - יוצאי דופן (מבחינת מוסכמת השמות): **`int`** המתאים ל **`Integer`** , ו- **`char`** המתאים ל- **`Character`**

- כל הטיפוסים העוטפים מקובעים (immutable)

- הטיפוסים העוטפים שימושיים כאשר יש צורך בעצם (למשל ביצירת אוספים של ערכים, ובשימוש בטיפוס גנרי)

Boxing and Unboxing

- ניתן לתרגם טיפוס פרימיטיבי לטיפוס העוטף שלו (boxing) ע"י קריאה לבנאי המתאים:

```
char pc = 'c';
Character rc = new Character(pc);
```

- ניתן לתרגם טיפוס עוטף לטיפוס הפרימיטיבי המתאים (unboxing) ע"י שימוש במתודות xxxValue המתאימות:

```
Float rf = new Float(3.0);
float pf = rf.floatValue();
```

- ג'אווה מאפשרת **מעבר אוטומטי** בין טיפוס פרימיטיבי לטיפוס העוטף שלו:

```
Integer i = 0;    // autoboxing
int n = i;        // autounboxing
if(n==i) // true
     i++;// i==1
System.out.println(i+n); // 1
```

בחזרה לשימוש בטיפוס גנרי

- נראה מחלקה שמשתמשת ב **Cell <T>** , שהיא אנלוגית למחלקה שהשתמשה ב **IntCell** :

```
public class TestGen {

    public static void main(String[] args) {
        Cell<Integer> x = null;
        Cell<Integer> y = new Cell<Integer>(5,x);
        y.printList();
        Cell<Integer> z = new Cell<Integer>(3,y);
        z.printList();
        z.setNext(new Cell <Integer>(2,y));
        z.printList();
        y.printList();
    }
}
```

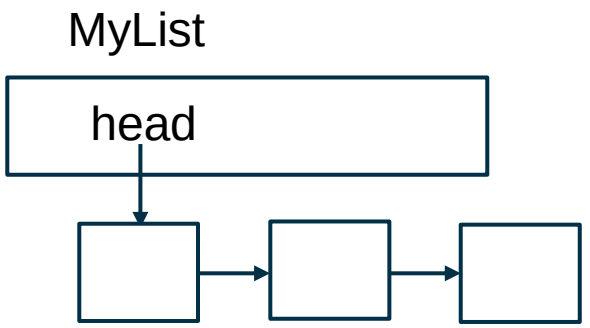
מי אתה Cell<T> ?

- האם **`Cell<T>`** באמת מייצג רשימה מקושרת?
- ב Java יש בשפה אמצעים טובים יותר להפשטת טיפוסים
- **`Cell`** אינו רשימה – הוא תא
  - ניתן (וצריך!) לבטא את שני הרעיונות **רשימה** ו**תא** כטיפוסים בשפה עם תכונות המתאימות לרמת ההפשטה שלהן
- נציג את המחלקה **`MyList<T>`** המייצגת רשימה

קירוב ראשון ל- MyList<T>

```
public class MyList <T> {

    private Cell <T> head;

    public MyList (Cell <T> head) {
              //code here
    }

    public Cell<T> getHead() {
              //code here
    }

    public void printList() {


       //code here

    }
}
```

המחלקה נקראת **MyList** ולא **List**
כדי שלא נתבלבל בינה ובין
java.util.List מהספרייה
הסטנדרטית של Java



קירוב ראשון ל- MyList<T>

המחלקה נקראת **MyList** ולא **List**
כדי שלא נתבלבל בינה ובין
java.util.List מהספרייה
הסטנדרטית של Java

```
public class MyList <T> {

    private Cell <T> head;

    public MyList (Cell <T> head) {
              this.head = head;
    }

    public Cell<T> getHead() {
              return head;
    }

    public void printList() {
              System.out.print("List: ");
              for (Cell <T> y = head; y != null; y = y.next())
                    System.out.print(y.cont() + " ");
              System.out.println();
    }
}
```

חסרונות המימוש

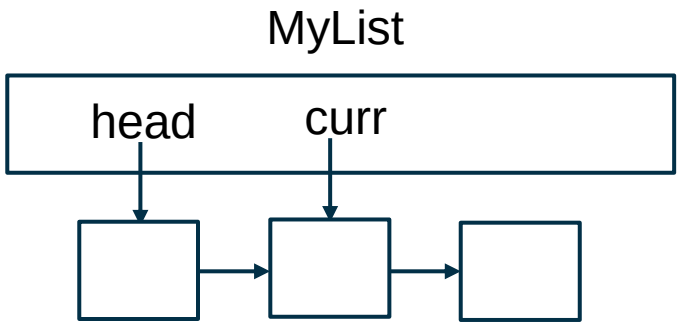
- מימוש הרשימה אמור להיות חלק **מהייצוג הפנימי** שלה **ומוסתר מהלקוח**
- במימוש המוצע **לקוחות** המחלקה **`MyList`** צריכים להכיר גם את המחלקה **`Cell`**

```
Cell<Integer> x = null;
Cell<Integer> y = new Cell<Integer>(5,x);
Cell<Integer> z = new Cell<Integer>(3,y);

MyList<Integer> l = new MyList<Integer>(z);
l.printList();
```

- הדבר פוגע **בהפשטת** רשימה מקושרת
  - למשל, אם בעתיד ירצה ספק **`Cell`** להחליף את המימוש לרשימה דו-כיוונית

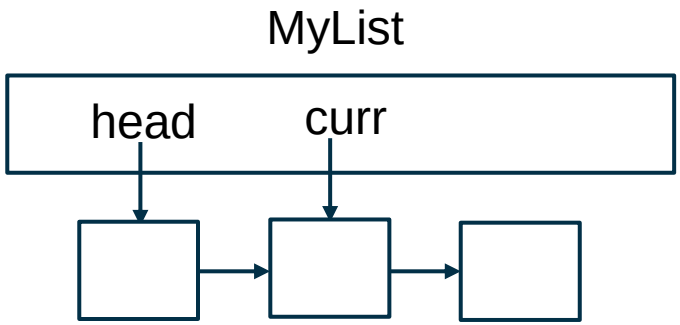
קירוב שני - MyList<T>

```
public class MyList<T> {

    private Cell <T> head;
    private Cell <T> curr;

    public MyList (T... elements) {


            //code here


    }

    public boolean atEnd(){
       //code here
    }

    /** @pre !atEnd() */
    public void advance() {
       //code here
    }
```



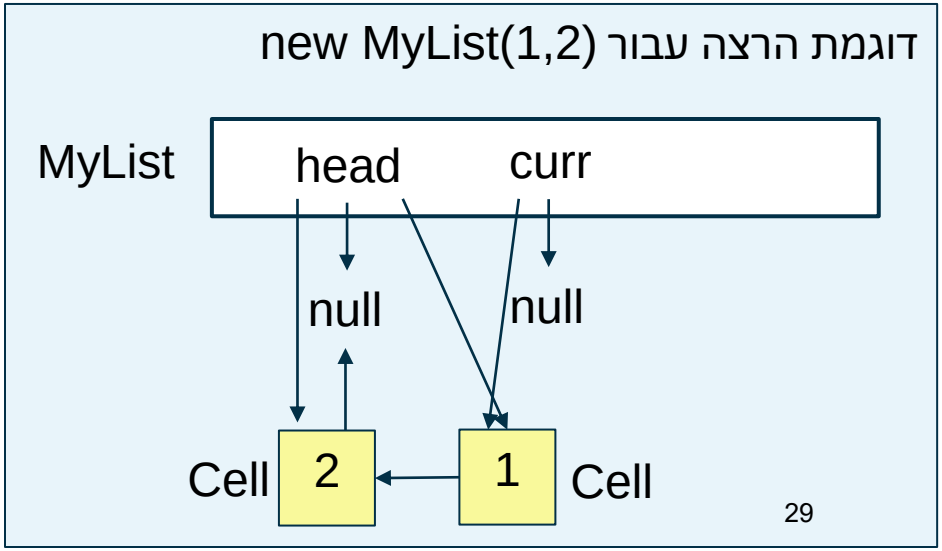
קירוב שני - MyList<T>

```
public class MyList<T> {

    private Cell <T> head;
    private Cell <T> curr;

    public MyList (T... elements) {


            //code here


    }

    public boolean atEnd(){
       return curr == null;
    }

    /** @pre !atEnd() */
    public void advance() {
       curr = curr.next();
    }
```



קירוב שני - MyList<T>

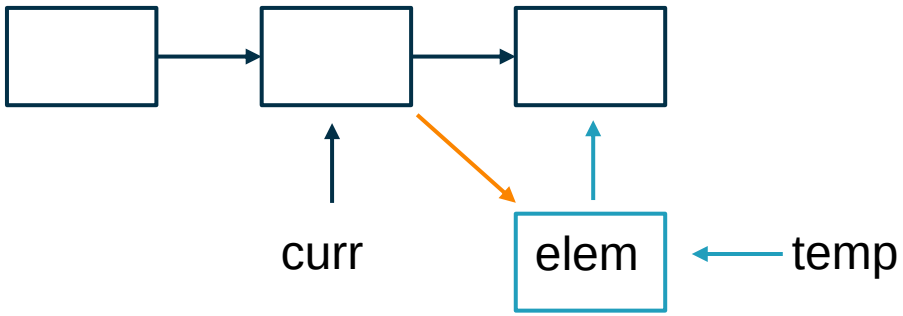
```
public class MyList<T> {

   private Cell <T> head;
   private Cell <T> curr;

   public MyList (T... elements) {
      this.head = null;
      for (int i = elements.length-1; i >= 0; i--) {
         head = new Cell<T>(elements[i], head);
      }
      curr = head;
   }

   public boolean atEnd(){
      return curr == null;
   }

   /** @pre !atEnd() */
   public void advance() {
      curr = curr.next();
   }
```



המשך - MyList<T>

```
/** @pre !atEnd() */
public T cont() {
   return curr.cont();
}

/** @pre !atEnd() */
public void addNext(T elem) {
   Cell<T> temp = new Cell<T>(elem, curr.next());
   curr.setNext(temp);
}
}
```

השרות אינו מחזיר את התא הנוכחי (טיפוס Cell) אלא את **התוכן** של התא הנוכחי (T)



המשך - MyList<T>

```
    /** @pre !atEnd() */
    public T cont() {
       return curr.cont();
    }

    /** @pre !atEnd() */
    public void addNext(T elem) {
       Cell<T> temp = new Cell<T>(elem, curr.next());
       curr.setNext(temp);
    }

    public void printList() {
       System.out.print("List: ");
       for (Cell <T> y = head; y != null; y = y.next())
          System.out.print(y.cont() + " ");
       System.out.println();
    }
}
```

ידפיס את תוצאת הפעלת השרות
toString של הטיפוס T על **y.cont()**

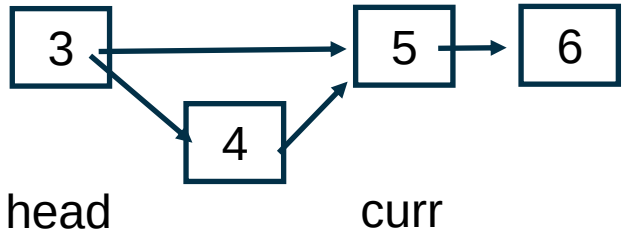
MyList<T>

- בעת לקוח הרשימה (**MyList**) אינו מודע לקיום מחלקת העזר **Cell<T>**:

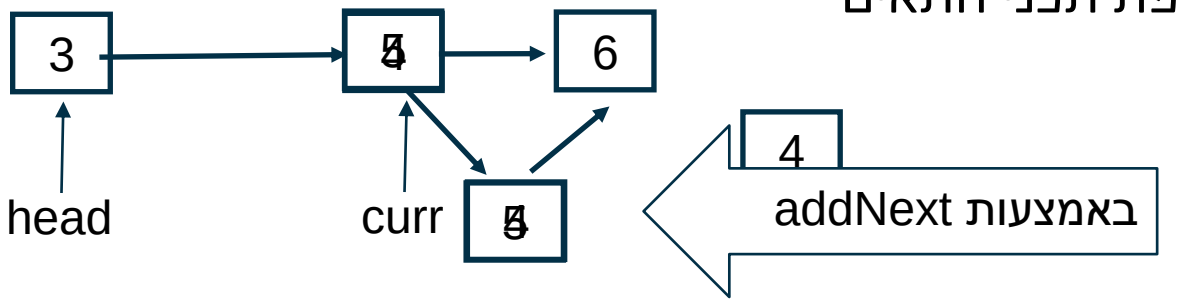
```
MyList<Integer> l = new MyList<Integer>(3,5);
l.printList();
l.advance();
l.addNext(4);
l.printList();
```

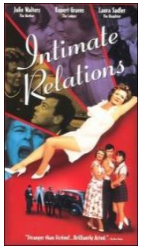
MyList<T>

- איך נממש את השרות **`addHere(int x)`** – שרות המוסיף את האיבר x למקום הנוכחי ברשימה:



- בשונה מהשרות **`addNext()`** אנו צריכים לשנות את ההצבעה לתא **`curr`**. לשם כך ניתן לנקוט כמה גישות:
  - **<u>גישה א'</u>**: תחזוקה של **`prev`** נוסף על **`curr`**
  - **<u>גישה ב'</u>**: נרוץ מתחילת הרשימה עד המקום אחד לפני הנוכחי (ע"י השוואת **`next()`** של כל תא ל **`curr`**)
  - **<u>גישה ג':</u>** החלפת תכני התאים





יחסים אינטימיים

- גישות א' ו- ב' פשוטות יותר רעיונית אך פחות אלגנטיות (תחזוקה, ביצועים)
- ננסה לממש את גישה ג'

```
    /** @pre !atEnd() */
public void addHere(T elem) {
    addNext(curr.cont);
    curr.cont = elem;
}
```

בעיה! השדה `cont` הוא שדה פרטי של `Cell`, ולכן לא נגיש ל `MyList`

- אולי במקרה זה דרישת הפרטיות של נראות של השדה `cont` היא מוגזמת?
  - הקלת הנראות של שדה אינה מוצדקת
  - ואולם, המחלקה `Cell<T>` היא **מחלקת עזר** של `MyList<T>` ולכן יש הצדקה למתן הרשאות גישה חריגות ל- `MyList<T>` לשדותיה הפרטיים של `Cell<T>`

- גם לו היתה ל `Cell` המתודה `setCont()` ניתן היה לומר כי לאור השימוש התכוף שעושה הרשימה בשרותי התא, ניתן היה **משיקולי יעילות** לאפשר לה גישה ישירה לשדה זה

יחסים אינטימיים ב Java

- אם **`MyList`** ו **`Cell`** באותה חבילה אפשר להשתמש בנראות חבילה - אבל אז כל מחלקה אחרת בחבילה תוכל גם היא לגשת לפריטים האלה של **`Cell`**

- ניתן להגדיר **אינטימיות** בין מחלקות ב Java ע"י הגדרת אחת המחלקות **כמחלקה פנימית** של המחלקה האחרת

- מחלקות פנימיות הן מבנה תחבירי בשפת Java המבטא **בין השאר** הכרות אינטימית

- הערה על דרגות נראות:
  - דרגת הנראות ב Java היא **ברמת המחלקה**. כלומר עצם מטיפוס כלשהו יכול לגשת גם לשדות הפרטיים של עצם אחר מאותו הטיפוס
  - ניתן היה לחשוב גם על נראות **ברמת העצם** (לא קיים ב Java)

התכנית לשבוע הקרוב

- מבנים מקושרים
- הכללה של המבנה ע"י הכללת טיפוסים (Generics)
- מחלקות פנימיות (או ייצוג "הכרות אינטימית" בשפת התכנות)
- הפשטת מעבר סידרתי על נתונים (איטרטורים)
- הספקת טיפוסים במשתנים מקומיים
- עוד על generics

Inner Classes

- מחלקה פנימית היא מחלקה שהוגדרה בתחום (Scope – בין הסוגריים המסולסלים) של מחלקה אחרת
- דוגמה:

```
public class House {
    private String address;

    public class Room {
        private double width;
        private double height;
    }
}
```

שימוש לב!

- Room אינה **שדה** של המחלקה House

Inner Classes

- מחלקה פנימית היא מחלקה שהוגדרה בתחום (Scope – בין הסוגריים המסולסלים) של מחלקה אחרת

- דוגמה:

```
public class House {
    private String address;
    private Room[] rooms;
    public class Room {
        private double width;
        private double height;
    }
}
```

שימוש לב!

- Room אינה **שדה** של המחלקה House
- אם רוצים ליצור שדה בזה יש לעשות זאת במפורש

מחלקות פנימיות

- הגדרת מחלקה כפנימית מרמזת על היחס בין המחלקה הפנימית והמחלקה העוטפת:
  - למחלקה הפנימית יש משמעות רק בהקשר של המחלקה החיצונית
  - למחלקה הפנימית יש הכרות אינטימית עם המחלקה החיצונית
  - המחלקה הפנימית היא מחלקת עזר של המחלקה החיצונית

- דוגמאות:
  - **`Collection`** ו- **`Iterator`**
  - **`Body`** ו- **`Brain`**
  - מבני נתונים המוגדרים ברקורסיה: **`List`** ו- **`Cell`**

סוגי מחלקות פנימיות

- ב Java כל מופע של עצם מטיפוס המחלקה הפנימית משויך לעצם מטיפוס המחלקה העוטפת

- השלכות
  - תחביר מיוחד לבנאי (פרטים בתרגול).
  - לעצם מטיפוס המחלקה הפנימית יש שדה הפנייה שמיוצר אוטומטית לעצם מהמחלקה העוטפת
  - כתוצאה מכך יש למחלקה הפנימית גישה לשדות ולשרותים (אפילו פרטיים!) של המחלקה העוטפת **ולהיפך**

סוגי מחלקות פנימיות

```
public class House {
    private String address;

    public class Room {
        private double width;
        private double height;

        public String toString(){
            return "Room " + address;
        }
    }
}
```

מחלקות פנימיות סטטיות

- ניתן להגדיר מחלקה פנימית כ **`static`** ובכך לציין שהיא אינה קשורה למופע מסוים של המחלקה העוטפת
- הדבר אנלוגי למחלקה שכל שרותיה הוגדרו כ **`static`** והיא משמשת כספריה עבור מחלקה מסוימת
- בשפת C++ יחס זה מושג ע"י הגדרת יחס **`friend`**

מחלקות פנימיות סטטיות

```
public class House {

    private String address;


    public static class Room {

        private double width;

        private double height;

        public String toString(){
            return "Room " + address;
        }

    }

}
```

מחלקות פנימיות בתוך מתודות

- ניתן להגדיר מחלקה פנימית בתוך מתודה של המחלקה החיצונית
- הדבר מגביל את תחום ההכרה של אותה מחלקה לתחום אותה המתודה בלבד.
- המחלקה הפנימית תוכל להשתמש במשתנים מקומיים של המתודה רק אם הם הוגדרו כ **`final`**
  - החל מ Java 8 ניתן לגשת גם למשתנים ש"מתנהגים" כמו משתנים שהוגדרו כ **`final`**, כלומר, מקבלים השמה פעם אחת בלבד לאורך חייהם (final effectively).

מחלקות פנימיות בתוך מתודות

```
public class Test {
     public void test(int num) {
          final int x = num+3;
          int y = num*2;
          int z= num-1;
          class Info{
               public String toString() {
                    return "**" + x + "**" + y + "**" + z;
               }
          }
          z = 4;
          System.out.println(new Info());
     }

     public static void main(String[] args) {
          Test t = new Test();
          t.test(5);
     }
}
```

האם `toString` יכולה לגשת גם ל num?

מה יקרה לקוד לאחר הוספת שורה זו?

מחלקות אנונימיות

- בעזרת מחלקות פנימיות ניתן להגדיר מחלקות אנונימיות – מחלקות ללא שם
- מחלקות אנונימיות שימושיות מאוד במערכות מונחות אירועים (כמו GUI)

הידור של מחלקות פנימיות

- המהדר (קומפיילר) יוצר קובץ class. עבור כל מחלקה. מחלקה פנימית אינה שונה במובן זה ממחלקה רגילה.

- שם המחלקה הפנימית **`Outer$Inner.class`**.

- אם המחלקה הפנימית אנונימית, שם המחלקה שיוצר הקומפיילר יהיה **`Outer$1.class`**.

חזרה ל MyList ו Cell

- בדי להסתיר מהלקוח של הרשימה את הייצוג הפנימי, ובדי לאפשר גישה לשדות הפרטיים של **`Cell`** נכתוב את **`Cell`** כמחלקה מקוננת, פרטית בתוך **`MyList`**

- האם מחלקה פנימית סטטית או לא?
  - אפשרות אחת: **`Cell`** אינה סטטית
    - אז כל עצם מסוג **`Cell`** משויך לעצם **`MyList`** כלומר לרשימה מסוימת, ומאפשר לעצם להכיר את הרשימה בה הוא מופיע.
    - אבל מה נעשה אם הוא יעבור לרשימה אחרת?
    - למעשה זה בלתי אפשרי! האבר (התוכן) יכול להיות מוכנס לרשימה אחרת, אבל לא העצם מטיפוס **`Cell`**
  - אפשרות שנייה: **`Cell`** סטטית
    - מה ההשלכות מבחינת הגנריות?

רשימה עם מחלקה מקוננת

- אם **`Cell`** מחלקה מקוננת לא סטטית בתוך **`MyList`** היא לא חייבת להיות מוגדרת בגנרית. טיפוס התוכן של ה **`Cell`** נקבע על פי הפרמטר האקטואלי של עצם ה **`MyList`** המתאים.

- בלומר הרשימה קובעת את סוג אבריה, וכל האברים שנוצרים עבור רשימה מסוימת שותפים לאותו סוג

- קצת יותר קל לכתוב את הקוד

- הערה: נראות השדות והשרותים של מחלקה מקוננת פרטית אינה משמעותית (בכל מקרה ידועים למחלקה העוטפת ורק לה).

```
public class MyList<T> {

    private class Cell {
        private T cont;
        private Cell next;

        public T cont() { return cont; }
        public Cell next() { return next;}
        // ...
    }

    private Cell head;
    private Cell curr;

    public MyList(...) { ... }

    public boolean atEnd() { return curr == null; }

    /** @pre !atEnd() */
    public void advance() { curr = curr.next(); }

    // ...
```

רשימה עם מחלקה מקוננת סטטית

- אם Cell סטטית היא חייבת להיות גנרית, כי אחרת, עבור:

```
private T cont;
```

נקבל הודעת שגיאה:

Cannot make a static reference to the non-static type T

- כי אם **`Cell`** סטטית, היא לא מתייחסת לעצם מטיפוס **`MyList`**, שטיפוס האבר שלו נקבע ביצירתו, אלא למחלקה **`MyList <T>`** שבה לא נקבע טיפוס קונקרטי ל **`T`**

- אם כן, מה הפרמטר הגנרי שלה? **`T`** או אחר?

- שתי האפשרויות הן חוקיות, אבל צריך להבין שבכל מקרה אלה שני משתנים שונים, והשימוש עלול להיות מבלבל

```
public class MyList<T> {

    private static class Cell<S> {
        private S cont;
        private Cell<S> next;

        public Cell(S cont, Cell<S> next) {
            this.cont = cont;
            this.next = next;
        }

        public S cont() { return cont; }
        public Cell<S> next() { return next;}
        // ...
    }

    private Cell<T> head;
    private Cell<T> curr;

    public MyList(/* ... */) { ... }

    public boolean atEnd() { return curr == null; }

    // ...
}
```

דיון: printList()

```
public void printList() {
    System.out.print("List: ");
    for (Cell <T> y = head; y != null; y = y.next())
            System.out.print(y.cont() + " ");
    System.out.println();
}
```

- **`printList()`** היא שרות גרוע
- **בעיה:** השרות פונה למסך – זוהי החלטה שיש לשמור "לזמן קונפיגורציה". אולי הלקוחה מעוניינת להדפיס את המידע למקום אחר
- **פתרון:** שימוש ב **`toString`** – שרות זה יחזיר את אברי הרשימה כמחרוזת והלקוחה תעשה במחרוזת כרצונה
- **בעיה:** השרות מכתיב את פורמט הדפסה (כותרות, רווחים, שורות חדשות) ומגביל את הלקוחה לפורמט זה. הלקוחה לא יכול לאסוף מידע זה בעצמה שכן הוא אפילו לא מכיר את המחלקה **`Cell`**

תוכנית לשני השיעורים הקרובים

- מבנים מקושרים
- הכללה של המבנה ע"י הכללת טיפוסים (Generics)
- מחלקות פנימיות (או ייצוג "הכרות אינטימית" בשפת התכנות)
- הפשטת מעבר סידרתי על נתונים (איטרטורים)
- הספקת טיפוסים במשתנים מקומיים
- עוד על generics

דיון: printList()

- אנחנו צריכים שלמחלקה MyList יהיה משהו שיודע להחזיר תשובה על שתי שאלות:
  1. מהו האיבר הבא?
  2. האם נותרו איברים נוספים?

```
public interface Iterator<E>{
        E next();
        boolean hasNext();
}
```

```
public interface Iterable<E>{
        Iterator<E> iterator();
}
```

```
public class MyList<E> implements Iterable<E>{
        …
        public Iterator<E> iterator() {// implementation }
}
```

אלגוריתם כללי להדפסת אוסף נתונים

- נדפיס את האיברים השמורים במבנה נתונים **collection** כלשהו:

```
for (Iterator iter = collection.iterator();
                                iter.hasNext( ); ) {
  System.out.println(iter.next());
}
```

גישה בעזרת משתנה
העזר לנתון וקידומו
לאיבר הבא

בדיקה:
האם גלשנו

הגדרת
משתנה עזר
ואתחולו

אלגוריתם כללי להדפסת אוסף נתונים

- נדפיס את האיברים השמורים במבנה נתונים **collection** כלשהו:

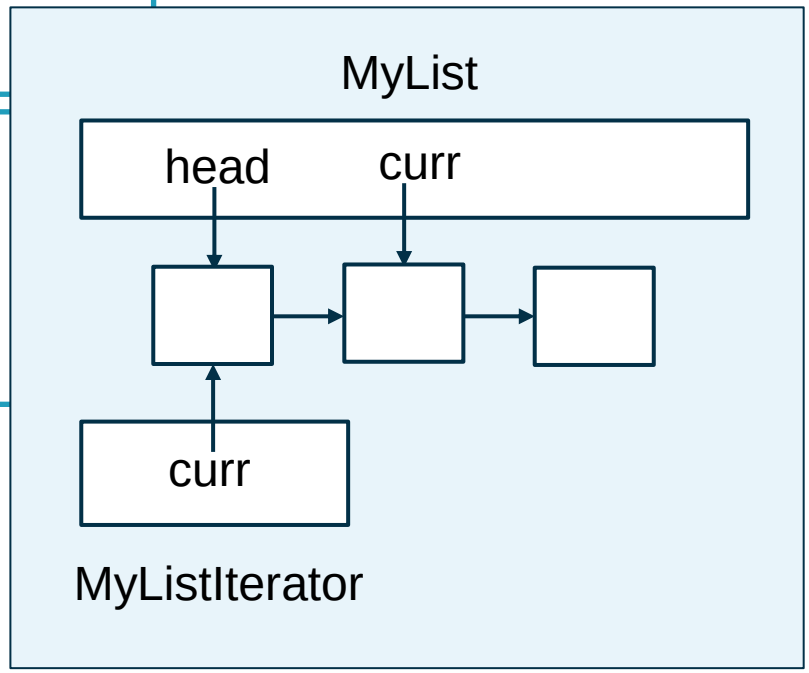
```
for (Iterator iter = collection.iterator();
                                    iter.hasNext( ); ) {
  System.out.println(iter.next());
}
```

- מבנה הנתונים עצמו אחראי לספק ללקוח איטרטור תיקני (עצם ממחלקה שממממשת את ממשק **Iterator**) המאותחל לתחילת מבנה הנתונים

- אם נרצה שהמחלקה **MyList** תספק ללקוחותיה את האפשרות לסרוק את כל האיברים ברשימה, עלינו לכתוב לה **Iterator**

תקני MyListIterator

```
class MyListIterator<S> implements Iterator<S> {

    private Cell<S> curr;

    public MyListIterator(Cell<S> cell) {
        this.curr = cell;
    }

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

    public S next() {
        S result = curr.getCont();
        curr = curr.getNext();
        return result;
    }

}
```



מספקת איטרטור ללקוחותיה **MyList<T>**

```
public class MyList<T> implements Iterable<T> {
  //...
  public Iterator<T> iterator() {
        return new MyListIterator<T>(head);
  }
}
```

- מחלקות המממשות את המתודה **iterator()** בעצם ממשות את המנשק **Iterable<T>** המכיל מתודה זו בלבד

- הצימוד בין **MyList** ו- **MyListIterator** חזק. על כן מקובל לממש את האיטרטור במחלקה פנימית של האוסף שעליו הוא פועל

- בעת הלקוח יכול לבצע פעולות על כל אברי הרשימה בלי לדעת מהו המבנה הפנימי שלה

printSquares

```
public void printSquares( Iterable<Integer> ds )
{
  for (Iterator<Integer> iter = ds.iterator();
       iter.hasNext();) {
       int i = iter.next();
       System.out.println(i*i);
  }
}
```

Autounboxing

What is the output for:

```
System.out.println(iter.next()*iter.next());
```

(שמרו לכן על הפרדה בין פקודות לשאילתות)

- הלקוח מדפיס את ריבועי אברי הרשימה בלי להשתמש בעובדה שזו אכן רשימה

- טיפוס הארגומנט `MyList<Integer>` יכול להיות מוחלף בשם המנשק `Iterable<Integer>`, ואז הלקוח לא ידע אפילו את **שמו** של טיפוס מבנה הנתונים

`for/in` (foreach)

- לולאת for שמבצעת את אותה פעולה על כל אברי אוסף נתונים כלשהו כה שכיחה, עד שב Java 5.0 הוסיפו אותה לשפה בתחביר מיוחד (**`for/in`**)
- הקוד מהשקף הקודם שקול לקוד הבא:

```
public void printSquares(MyList<Integer> list) {
  for (int i : list)
        System.out.println(i*i);
}
```

- יש לקרוא זאת כך:
  - "לכל איבר **`i`** מטיפוס **`int`** שבאוסף הנתונים **`list`**..."
- אוסף הנתונים **`list`** חייב לממש את המנשק **`Iterable`**

Iterator

- נממש איטרטור פשוט שעובד על מערך של שלמים.
- מטרת האיטרטור – בהנתן חסם תחתון lowerBound, האיטרטור יחזיר רק את האיברים שגדולים או שווים לו.
- אבחנות:
  - אם במערך יש k איברים, יכול להיות שהאיטרטור יחזיר פחות מ k איברים.
  - החלק המורכב – כיצד נדע שאין יותר איברים הגדולים או שווים ל lowerBound?
- אחרי מימוש האיטרטור נקבל נעשה בו שימוש באופן הבא:

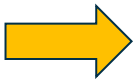
```
int[] arr = {1,4,7,3,6,2};
Iterator<Integer> it = new MyIterator(arr, 2);
while(it.hasNext()){
        System.out.println(it.next());
}
```

```
public class MyIterator implements Iterator<Integer>{
	private int[] arr;
	private int lowerBound = 0;
	private int currPos = 0;
	private int lastValidIndex = -1;

	public MyIterator(int[] arr, int lowerBound){
	  this.arr = arr;
	  this.lowerBound = lowerBound;
	  for (int i = arr.length-1; i>= 0; i--){
		if (arr[i] >= lowerBound){
				this.lastValidIndex = i;
					break;
		}
	  }}

	public boolean hasNext() { return currPos <= lastValidIndex; }

	public Integer next() {
	  while(arr[currPos] < lowerBound){
				currPos++;
	  }
	  return arr[currPos++];
	}
}
```

```
int[] arr = {1,4,7,3,6,1};
Iterator<Integer> it = new MyIterator(arr, 2);
while(it.hasNext()){
	System.out.println(it.next());
}
```

```
public class MyIterator implements Iterator<Integer>{
        private int[] arr;
        private int lowerBound = 0;
        private int currPos = 0;
        private int lastValidIndex = -1;

        public MyIterator(int[] arr, int lowerBound){
          this.arr = arr;
          this.lowerBound = lowerBound;
          for (int i = arr.length-1; i>= 0; i--){
                  if (arr[i] >= lowerBound){
                                  this.lastValidIndex = i;
                                        break;
                  }
          }}

        public boolean hasNext() { return currPos <= lastValidIndex; }

        public Integer next() {
          while(arr[currPos] < lowerBound){
                                  currPos++;
          }
          return arr[currPos++];
        }
}
```

```
it.arr = {1,4,7,3,6,1}
it.lowerBound = 2
it.currPos = 0
it.lastValidIndex = 4
                  //index of 6
```



```
int[] arr = {1,4,7,3,6,1};
Iterator<Integer> it = new MyIterator(arr, 2);
while(it.hasNext()){
        System.out.println(it.next());
}
```

```
public class MyIterator implements Iterator<Integer>{
        private int[] arr;
        private int lowerBound = 0;
        private int currPos = 0;
        private int lastValidIndex = -1;
```

```
it.arr = {1,4,7,3,6,1}
it.lowerBound = 2
it.currPos = 0
it.lastValidIndex = 4
                //index of 6
```

```
        public boolean hasNext() { return currPos <= lastValidIndex; }

        public Integer next() {
          while(arr[currPos] < lowerBound){
                                currPos++;
          }
          return arr[currPos++];
        }
}
```

```
int[] arr = {1,4,7,3,6,1};
Iterator<Integer> it = new MyIterator(arr, 2);
while(it.hasNext()){
        System.out.println(it.next());
}
```

```
public class MyIterator implements Iterator<Integer>{
    private int[] arr;
    private int lowerBound = 0;
    private int currPos = 0;
    private int lastValidIndex = -1;

    public boolean hasNext() { return currPos <= lastValidIndex; }

    public Integer next() {
      while(arr[currPos] < lowerBound){
                        currPos++;
      }
      return arr[currPos++];
    }
}
```

it.arr = {1,4,7,3,6,1}
it.lowerBound = 2
it.currPos = 2
it.lastValidIndex = 4
//index of 6

arr[(arr[1] < 2 ???

return arr[1]

```
int[] arr = {1,4,7,3,6,1};
Iterator<Integer> it = new MyIterator(arr, 2);
while(it.hasNext()){
        System.out.println(it.next());
}
```

התכנית לשבוע הקרוב

- מבנים מקושרים
- הכללה של המבנה ע"י הכללת טיפוסים (Generics)
- מחלקות פנימיות (או ייצוג "הכרות אינטימית" בשפת התכנות)
- הפשטת מעבר סידרתי על נתונים (איטרטורים)
- הספקת טיפוסים במשתנים מקומיים
- עוד על generics

הסקת טיפוסים

- המטרה – במקרים מסוימים, ניתן לוותר על הצהרת הטיפוס הסטטי של משתנה מקומי. קיים החל מ Java 10

```
List<String> cities = List.of("Brussels", "Cardiff",
"Cambridge");

Map<String, Integer> citiesPopulation
        = Map.ofEntries(Map.entry("Brussels", 1_139_000),
                        Map.entry("Cardiff", 341_000));
```

הסקת טיפוסים

- המטרה – במקרים מסוימים, ניתן לוותר על הצהרת הטיפוס הסטטי של משתנה מקומי. קיים החל מ Java 10

```
var cities = List.of("Brussels", "Cardiff",
"Cambridge");

var citiesPopulation
        = Map.ofEntries(Map.entry("Brussels", 1_139_000),
                        Map.entry("Cardiff", 341_000));
```

הסקת טיפוסים - מגבלות

- ניתן להשתמש ב var רק עבור משתנים שקיים עבורם אתחול בגוף ההצהרה על המשתנה

```
var i;
```

- משתנה מטיפוס var צריך להיות מאותחל כך שיהיה ניתן להסיק מהאתחול את הטיפוס:

```
var i = null;
```

- לא ניתן להגדיר פרמטר לפונקציה מטיפוס var (למה?)

הסקת טיפוסים - מגבלות

- הטיפוס נקבע לפי האתחול של המשתנה

```
var lst = new PolarPoint(…); //lst is of type PolarPoint
```

- לא ניתן להשתמש ב var כהצבה לפרמטר גנרי

```
List<var> i = new ArrayList<String>();
```

- כן ניתן להשתמש ב var בשביל טיפוסים גנריים, אבל בזהירות!

```
var strArrList = new ArrayList<String>();
        //strArrList is of type ArrayList<String>
var objArrList = new ArrayList<>();
        //objArrList is of type ArrayList<Object>
```

התכנית לשבוע הקרוב

- מבנים מקושרים
- הכללה של המבנה ע"י הכללת טיפוסים (Generics)
- מחלקות פנימיות (או ייצוג "הכרות אינטימית" בשפת התכנות)
- הפשטת מעבר סידרתי על נתונים (איטרטורים)
- הספקת טיפוסים במשתנים מקומיים
- עוד על generics

Diamond Syntax

- הכתיב הבא חוקי ב Java:

```
List<String> sArrList = new ArrayList<>();
```

ושקול ל:

```
List<String> sArrList = new ArrayList<String>();
```

כשיוצרים מופע של טיפוס גנרי, ה שמצד ימין של אופרטור ההשמה (=) חוזר על ערך פרמטר הטיפוס. Java מאפשרת לנו לקצר ולהימנע מחזרה זו, והקומפיילר למעשה יסיק את ערך פרמטר הטיפוס לפי מה שמופיע משמאל לאופרטור ההשמה.

מה עושים ללא מחלקות גנריות

- אחת הדוגמאות השכיחות לשימוש בהמרת טיפוסים ב Java היא השימוש במבני נתונים לפני Java 1.5 (קרויה גם Java 5)

- מכיוון שעד לגרסה 1.5 לא ניתן היה להשתמש בטיפוסים מוכללים (generics), נאלצו כותבי הספריות להניח שהאברים הם מהמחלקה הכללית ביותר, כלומר Object

- נניח כי רוצים לכתוב מנשק ו/או מחלקה עבור מחסנית, שתאפשר ליצור מחסנית של שלמים, מחסנית של מחרוזות, וכו' **ללא שימוש ב Generics**

- בדוגמה – מנשק למחסנית, ומחלקה ממשת (ללא החוזה)

מנשק מחסנית

```
interface Stack {
  public Object top ();
  public void push(Object t);
  public void pop();
  public boolean empty();
  public boolean full();
}
```

מימוש מחסנית פשוט

```
public class FixedCapacityStack implements Stack{

   private Object [] content;
   private int capacity;
   private int topIndex;

   public FixedCapacityStack(int capacity){
            content = new Object[capacity];
            this.capacity = capacity;
            topIndex = -1;
   }

   public Object top () {
            return content[topIndex];
    }
```

מימוש מחסנית פשוט

```
    public void push(Object t) {
            content[++topIndex] = t;
    }

    public void pop() {
            topIndex--;
    }

    public boolean empty() {
            return (topIndex < 0);
    }

    public boolean full() {
            return (topIndex >= capacity - 1) ;
    }
}
```

איך נשתמש במחסנית?

- נניח שרוצים מחסנית של מחרוזות:

```
Stack s = new FixedCapacityStack(5);
s.push("hello");
String t1 = s.top();                    // compilation error
String t2 = (String) s.top();           //ok
```

- באחריות **המתכנתת** לוודא שכל האברים המוכנסים למחסנית הם מאותו טיפוס (כאן מחרוזות), אחרת ה Casting יכשל.

```
Stack s = new FixedCapacityStack(5);
s.push("hello");
s.push(new Integer(4));
s.push(new PolarPoint(3,2));
String t2 = (String) s.top();           //compilation ok. Runtime Error !
```

בטיחות טיפוסים

- מכיוון שבדיקת ההמרה נעשית בזמן ריצה אנחנו מאבדים בטיחות טיפוסים
- זהו דבר שאינו רצוי – אנו מעוניינים להעביר בדיקות רבות ככל הניתן לזמן קומפילציה
  - מדוע?
- פתרון אחר: מנשק/מחלקה נפרדת לכל טיפוס איבר – שכפול קוד!
- הוספת הטיפוסים המוכללים לשפה פותרת גם את בעיית בטיחות הטיפוסים וגם את בעיית שכפול הקוד

מחלקה מוכללת (גנרית)

- מנגנון ההכללה מיועד לאפשר שימוש חוזר בקוד בלי לאבד מידע לגבי הטיפוס הסטאטי של עצם
- בלי הכללה, שימוש חוזר בקוד מתבצע על ידי השמת התייחסות מטיפוס אחד לטיפוס אחר, יותר כללי; מאותו רגע אין דרך לשחזר את הטיפוס הסטאטי המקורי בלי המרה
- תפקיד ההכללה הוא למנוע צורך בהמרות, שנבדקות מאוחר
- אבל העניינים מסתבכים בגלל האינטראקציה בין מנגנון ההכללה ובין יחס ההורשה (יחס ה-is-a)
- קושי נוסף: תאימות בין גרסאות גנריות ולא גנריות

פתרון שמשתמש ב Generics

- נגדיר את המנשק הגנרי Stack<T>

```
interface Stack<T> {
    public T top ();
    public void push(T t);
    …
}
```

- ואת המחלקה הגנרית FCStack<T> שממשת אותו:

```
class FCStack <T> implements Stack <T> {…}
```

- וכך נוכל לייצר מחסנית של מחרוזות , למשל:

```
Stack <String> vs = new FCStack <String>();
```

בטיחות טיפוסים

```
Stack <String>  ss = new FCStack <String> (5);
ss.push("The letter A");
ss.push(new Integer(3));
String t = ss.top();  // same as:(String)ss.top();
```

- מכיוון שרק מחרוזות יכולות להיות מוכלות במחסנית אין עוד צורך בהמרה

```
Stack <Rectangle>  sr = new FCStack <Rectangle>(5);
Rectangle rr = new Rectangle(...)
Rectangle rc = new ColoredRectangle(...)
ColoredRectangle cc = new ColoredRectangle(...)

sr.push(rr);
sr.push(rc);
sr.push(cc);
```

למה טוב שהקומפיילר שומר?

```
List names = new ArrayList(); //warning: raw type
names.add("Kyle");
names.add("Eric");
names.add(Boolean.FALSE);
```

```
for (Object o : names){
        String s = (String)o;
        System.out.println(s.toUpperCase());
}
//throws ClassCastException
//    java.lang.Boolean cannot be cast to ja
```

גילוי השגיאה
בזמן קומפילציה
ולא בזמן ריצה!

```
List<String> names = new ArrayList<>();
names.add("Kyle");
names.add("Eric");
names.add(Boolean.FALSE); //compilation error!
```

טיפוסים נאים (raw types)

- מנגנון ההכללה נוסף לג'אווה מאוחר, ולכן היה צורך לאפשר שימוש במחלקות פרמטריות גם מקוד ישן שאין בו הכללות.

```
Stack <String> vs = new FCStack <String>();

Stack raw =  new FCStack(); //What about T?
```

- ההוספה של טיפוסים גנריים לשפה ב Java 5 נאלצה להתחשב בנושא התאימות לאחור. בלומר, Java הגדירה מערכת טיפוסים שמאפשרת לקוד הבא להיות חוקי:

```
raw = vs;  // ok, for backward compatibiity
vs = raw;  // compiler warning: "unckecked" conversion
```

מנגנון מחיקת טיפוסים (Type Erasure)

- בלומר, FCStack ו- FCStack<String> הינם טיפוסים תואמים, במידה מסוימת
- זה מרמז על תכנון מערכת הטיפוסים של Java החל מ Java 5 (לאחר הוספת הטיפוסים הגנריים לשפה) והטיפול בטיפוסים הגנריים
- מנגנון זה נקרא מנגנון מחיקת הטיפוסים (Type Erasure)

איך זה עובד?

- הקומפיילר ממפה את כל המחלקות המובללות `FCStack<Something>` למחלקה אחת רגילה (לא מובללת) שלמעשה תואמת למחלקה `FCStack<Object>`

- בקוד שמשתמש במחלקה מובללת, **הקומפיילר מוסיף לקוד המרות** על מנת לבצע השמות מ-`Object` לטיפוס הספציפי, למשל String

- הקומפיילר מוודא שההמרה תמיד תצליח ולעולם לא תודיע על `ClassCastError`:

```
String t = (String) s.top();
```

- בלומר, הטיפוס המוכלל (T) **נמחק** מהקוד שהקומפיילר מייצר; הוא שימושי רק לבדיקות תקינות טיפוסים בזמן קומפילציה; התהליך נקרא מחיקה (erasure)

מנגנון מחיקת הטיפוסים

```java
public class Gen<T> {
    T t;
    Gen<T> gen;

    public Gen(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }

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

קודם כל, נטפל בהגדרת המחלקה Gen.
בזמן קומפילציה הטיפוס הגנרי מוחלף **בגבול העליון**.
מהו הגבול העליון? נגדיר זאת בהמשך, בדוגמה זו מדובר ב **Object** , כי לא צוין אחרת בהגדרת הטיפוס הגנרי

```java
public static void main(String[] args){
        Gen<String> b = new Gen<>("abc");
        String item = b.getT();
}
```

מנגנון מחיקת הטיפוסים

```java
public class Gen<T> {
    T t;
    Gen<T> gen;

    public Gen(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }

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

```java
public class Gen {
    Object t;
    Gen gen;

    public Gen(Object t) {
        this.t = t;
    }

    public Object getT() {
        return t;
    }

    public void setT(Object t) {
        this.t = t;
    }
}
```

הגדרת המחלקה לפני ואחרי קומפילציה

מנגנון מחיקת הטיפוסים

```
public static void main(String[] args){
    Gen<String> b = new Gen<>("abc");
    String item = b.getT();
}
```

```
public static void main(String[] args){
    Gen b = new Gen("abc");
    String item = (String)b.getT();
}
```

הקוד שעושה שימוש במחלקה הגנרית לפני ואחרי הקומפילציה

מנגנון מחיקת הטיפוסים

```
public class Gen<T> {
    T t;
    Gen<T> gen;

    public Gen(T t) {
        this.t = t;
    }

    public T getT() {
        return t;
    }

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

התייחסות מיוחדת:

```
public class SGen extends Gen<String> {
    public SGen(String t) {
        super(t);
    }

    public void setT(String t) {
        System.out.println(t);
        super.setT(t);
    }
}
```

הקוד המקורי לפני קומפילציה

מנגנון מחיקת הטיפוסים

```
public class Gen {
    Object t;
    Gen gen;

    public Gen(Object t) {
        this.t = t;
    }

    public Object getT() {
        return t;
    }

    public void setT(Object t) {
        this.t = t;
    }
}
```

התייחסות מיוחדת:

```
public class SGen extends Gen{
    public SGen(String t) {
        super(t);
    }

    public void setT(String t) {
        System.out.println(t);
        super.setT(t);
    }
}
```

בעיה!
setT של SGen לא דורסת את setT של Gen.

מנגנון מחיקת הטיפוסים

```
public class Gen {
    Object t;
    Gen gen;

    public Gen(Object t) {
        this.t = t;
    }

    public Object getT() {
        return t;
    }

    public void setT(Object t) {
        this.t = t;
    }
}
```

הקומפיילר מייצר פונקציית גשר:

```
public class SGen extends Gen{
    public SGen(String t) {
        super(t);
    }

    public void setT(Object t) {
        setT((String)t);
    }

    public void setT(String t) {
        System.out.println(t);
        super.setT(t);
    }
}
```

הכללה ויחס is-a

```
Stack <String>  ts = new FCStack <String> (5);
Stack <Object>  to = new FCStack <Object> (5);
to = ts;
ts.push("The letter A");
ts.push(new Integer(3));
to.push(new Integer(3));
```

- מסקנה: **FCStack<String>** **<u>אינו</u>** סוג של **FCStack<Object>**
- זה לא אינטואיטיבי אבל נכון

Covariance

- בימים הראשונים של Java, הוחלט שההשמה הבאה היא חוקית:

```
String [] strArr = new String[5];
Object [] objArr = strArr;
```

(הסיבה היא ש String מקיימת עם Object יחס is-a)

- בלומר, עבור מערכים: אם ערך מטיפוס C יכול להיות מושם למשתנה מטיפוס P אז ערך מטיפוס C[ ] יכול להיות מושם לתוך משתנה מטיפוס P[ ]
- תכונה זו נקראת Covariance: מערכים הם Convariant

- שימו לב כי ההשמה הנ"ל חוקית מבחינה תחבירית בלבד. `objArr` מצביע למערך של מחרוזות, ולכן שימוש שגוי בו יגרום לשגיאת זמן ריצה.

```
objArr[0] = new Integer(12); // throws ArrayStoreException
```

Covariance vs. Invariance

- Covariance של מערכים נחשב ל misfeature של השפה, וההמלצה היא שלא להשתמש בו כשכותבים קוד חדש
- האם התנהגות דומה מתקבלת גם עבור טיפוסים גנריים? כלומר, האם הקוד הבא יעבור קומפילציה?

```
FCStack <Object>  to = new FCStack <String> (5);
```

- לא יעבור קומפילציה!
- `FCStack< String >` אינו מקיים יחס is-a עם `FCStack< Object >`
- תכונה זו נקראת Invariance: טיפוסים גנריים הם Invariant

הגבול הוא השמיים

- **גבול עליון** הוא שם של המחלקה או המנשק שממנה יורש הטיפוס הפרמטרי
- כאשר הגבול העליון הוא Object לא ניתן לבצע כל פעולה על עצמים מהטיפוס הגנרי
- על כן, בהגדרת טיפוס גנרי ניתן לספק גבול עליון אחר
- שימו לב: בשימוש בטיפוס נא, פרמטר הטיפוס מוחלף ב**"גבול העליון"** (בדרך כלל Object)

- הדבר מאפשר להשתמש בגוף המחלקה הגנרית בשירותים המוגדרים באותו גבול עליון ללא צורך בהמרה

```
public class SortedSetImplementation<T extends Comparable>  {
   ...
   T elem1 = ...
   T elem2 = ...
   elem1.compareTo( elem2)  ....
   expectComparable(elem1);  //elem1 is indeed Comparable
}
```

מוזרויות

- בגלל שבג'אווה הכללה ממומשת באמצעות **מנגנון המחיקה**, בזמן ריצה אין זכר לפרמטר הטיפוס

- כלומר, בזמן ריצה אי אפשר להבחין בין עצם מטיפוס **`FCStack<String>`** ובין עצם מטיפוס **`FCStack<Integer>`**, ובפרט, בזמן ריצה נראה ששניהם מאותה מחלקה

- זה משפיע על בדיקת שייכות למחלקה (**`instanceof`**), על המרות של עצמים מובללים, ועל שדות המסומנים **`static`**

- וזה מונע אפשרות לקרוא לבנאי על פי פרמטר טיפוס, כלומר:

```
<T> void m(T x) { T y = new T(); ...} // illegal
```

- ויש עוד הרבה מזה...

למשל...

- רצינו לשלב את הקוד הבא (שמצאנו בגרסה ישנה של המוצר) במוצר החדש:

```
public static void printList(List list) {
        for(int i=0, n=list.size(); i < n; i++) {
                if (i % 2 == 0) {
                        System.out.println(list.get(i));
                }
        }
}
```

- בדי להימנע מאזהרות קומפילציה נשנה את **List** לטיפוס מוכלל:

```
public static void printList(List<Object> list) {
        for(int i=0, n=list.size(); i < n; i++) {
                if (i % 2 == 0) {
                        System.out.println(list.get(i));
                }
        }
}
```

- לא טוב, לא ניתן להעביר לשרות **List<String>**

ג'וקרים

- נשתמש בג'וקר (סימן שאלה - ?)

```
public static void printList(List<?> list) {
        for(int i=0, n=list.size(); i < n; i++) {
                if (i % 2 == 0) {
                        Object obj = list.get(i);
                        System.out.println(obj);
                }
        }
}
```

ג'וקרים

- בדי שנוכל **לבצע פעולות** ספציפיות על אברי הרשימה יש לספק **חסם עליון**, כמו בשרות:

```
public static double sumPerimiters(List<? extends IShape> list) {
        double total = 0.0;
        for(IShape n : list)
                total += n.perimeter();
        return total;
}
```

- משמעות ההגדרה: הטיפוס הגנרי של list הוא טיפוס המרחיב את IShape, כולל IShape עצמו כמובן.
  - שימו לב לשימוש ב extends גם עבור מנשקים. זהו תחביר מיוחד להרחבות.
- שימוש בשירות:

```
List<IShape> shapes = ...
List<Circle> circles = ...
List<Triangle> triangles = ...
double shapesPerimeterSum = sumPerimiters(shapes);
double circlesPerimeterSum = sumPerimiters(circles);
double trianglesPerimeterSum = sumPerimiters(triangles);
```

ג'וקרים

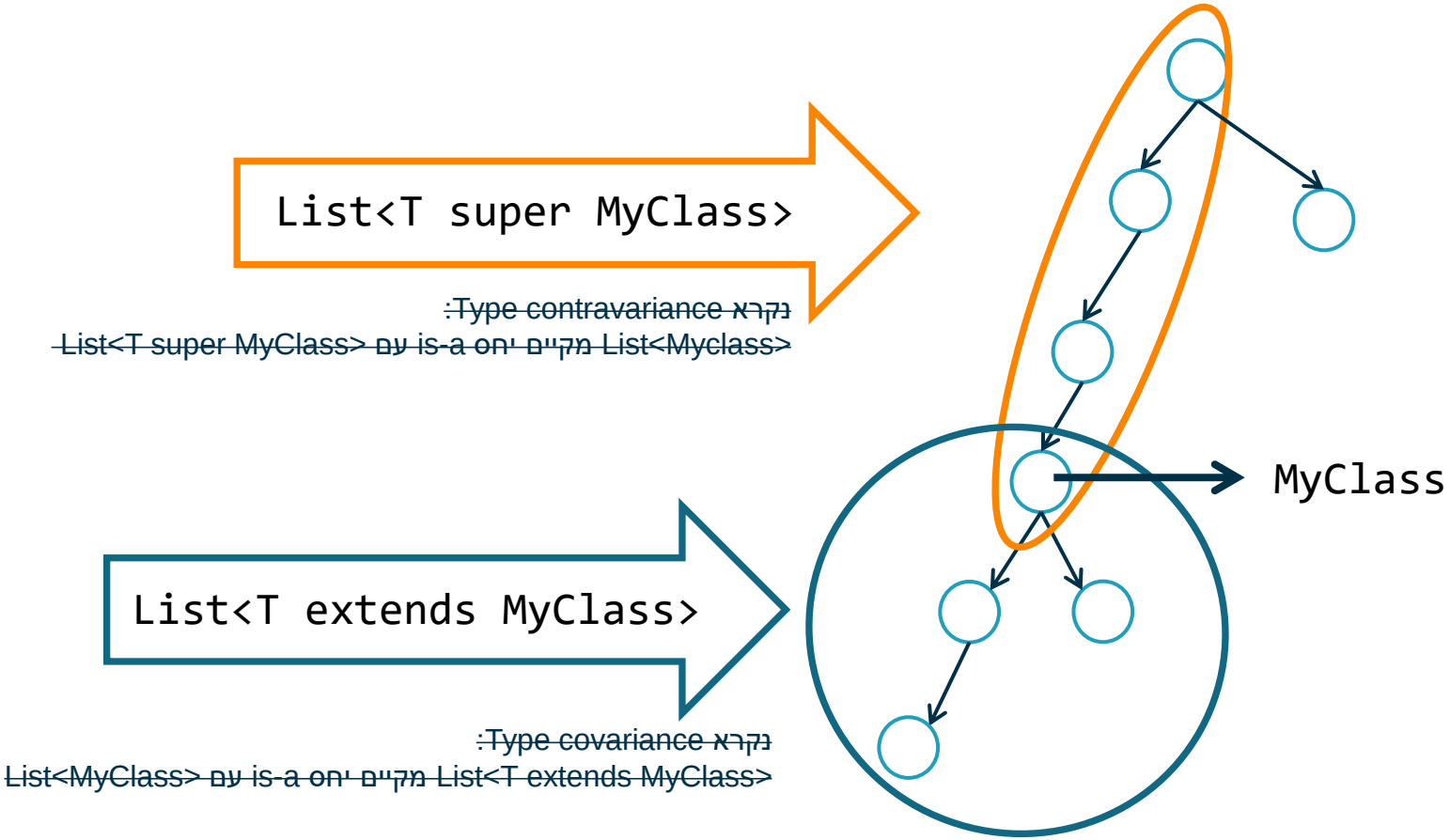
- יש גם חסמים תחתונים:

```
public static boolean addItem(List<? super ColoredRectangle> lst,
                              ColoredRectangle item) {
        return lst.add(item);
}
```

- המשמעות: הטיפוס הגנרי של הרשימה list הוא ColoredRectangle או טיפוס שאותו ColoredRectangle מרחיב.
- שימוש בשירות:

```
List<ColoredRectangle> cRectangles=...;
List<Rectangle> rectangles=...;
List<Object> objects=...;
ColoredRectangle cRect=...;
addItem(cRectangles, cRect);
addItem(rectangles, cRect);
addItem(objects, cRect);
```

super vs. extends



שירותים מוכללים

- ניתן להגדיר טיפוס גנרי עבור שירות בודד, ולא רק למחלקה

```
public class Test{
    public static void main(String[] args){
       List<Integer> intLst = Arrays.asList(1,2,3);
       List<String> strLst = Arrays.asList("a", "b", "c");
       int firstInt = getFirstItem(intLst);
       String firstStr = getFirstItem(strLst);
    }


    /*
     * @pre list.size() > 0
     */
    public static <T> T getFirstItem(List<T> list){
       return list.get(0);
    }
}
```

השירות getFirstItem מכיל פרמטר גנרי <T>.
ערכו של הפרמטר הגנרי יקבע בזמן הקריאה לשירות. למשל, כאשר נשלח List<String> כפרמטר, ערכו של T יקבע ל String.

שירותים מוכללים

- האם השירותים הבאים זהים?

```
Public static <T> void f1(List<T> l1, List<T> l2) { }

Public static void f2(List<?> l1, List<?> l2) {}
```

```
public static void main(String[] args) {
        List<String> l1 = null;
        List<Integer> l2 = null;
    ✗   f1(l1, l2);
    ✓   f2(l1, l2);
}
```

ובהקשר של מחלקות פנימיות...

```
public class MyType<E>{

     class Inner{}
     static class Nested{}

     public static void main(String[] args) {
          MyType mt; //warning: MyType is a raw type
          MyType.Inner inn; //warning: MyType.Inner is a raw type
          MyType.Nested nest; //no warning, not a parametrized type
          MyType<Object> mt1; //no warning
          MyType<?> mt2; //no warning, ? is OK for a type

     }

}
```

Raw שונה מ <Object>

```
public static void main(String[] args){
     List<String> strLst = new ArrayList<>();
     appendNewObject(strLst); //compilation error!
     for (String s: strLst) {
          System.out.println(s);
     }
}

public static void appendNewObject(List<Object> lst){
     lst.add(new Object());
}
```

מה היה קורה אם הפונקציה appendNewObject היתה מקבלת List נא?

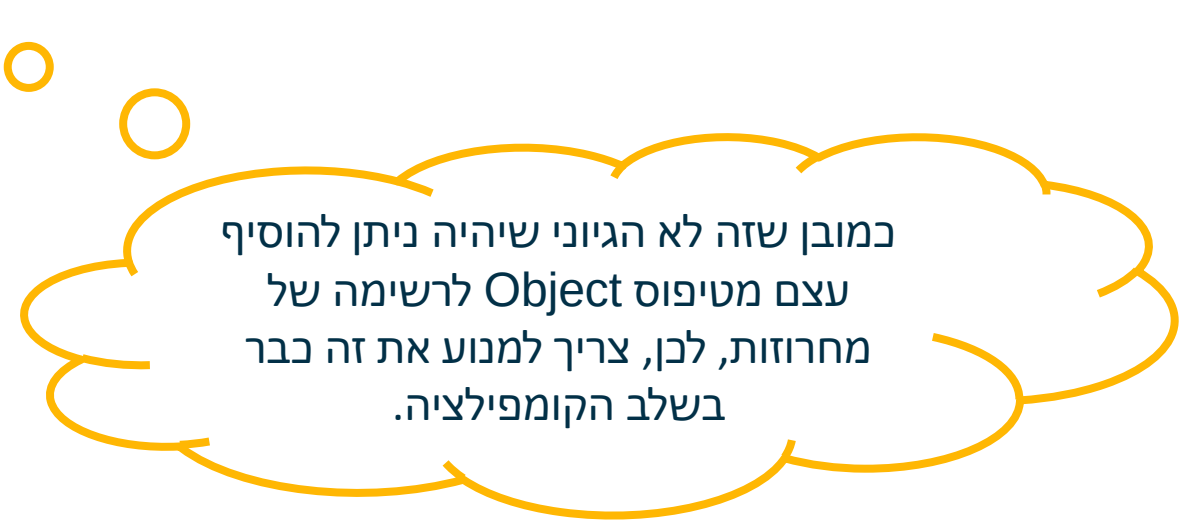
Raw שונה מ <Object>

```
public static void main(String[] args){
    List<String> strLst = new ArrayList<>();
    appendNewObject(strLst);  //succeeds!
    for (String s: strLst) {
        System.out.println(s);
    }
}

public static void appendNewObject(List lst){
    lst.add(new Object());
}
```

אם הפונקציה appendNewObject היתה מקבלת List נא, הקוד היה מתקמפל.
אבל היתה מתקבלת שגיאת זמן ריצה מסוג ClassCastException על השורה המסומנת בחץ, כי הלולאה מנסה להמיר כל איבר ברשימה לטיפוס String אבל הוכנס לתובה עצם מטיפוס Object.

Raw שונה מ <?> (wildcard)

```
public static void main(String[] args){
    List<String> strLst = new ArrayList<>();
    appendNewObject(strLst); //this is fine

}
public static void appendNewObject(List<?> lst){
    lst.add(new Object()); //compilation error!
}
```



במחלקת בסיס <?>

```
public static void printCollection(Collection<?> c){
    for (Object o: c){
        System.out.println(o);
    }
}
```

- ניתן לשלוח לפונקציה printCollection כל אוסף.
- בתוך printCollection ניתן לגשת לאלמנטים מתוך c ולשייך להם את הטיפוס הסטטי Object.

```
Collection<?> c = new ArrayList<>();
c.add(new Object()); // Compile time error
```

סיכום generics

- מנגנון ההכללה מאפשר להימנע מהמרות בלי לשכפל קוד
- קוד שאין בו המרות מפורשות ושאין בו טיפוסים נאים (ליתר דיוק, אם הקומפיילר לא הזהיר לגבי השימוש בטיפוסים נאים) הוא בטוח מבחינת טיפוסים (type safe)
- קוד כזה לא יכשל בביצוע המרה בזמן ריצה: הבדיקות מועברות לזמן הקומפילציה
- השימוש בהכללה מסבך הצהרות על טיפוסים בגלל האינטראקציה הלא אינטואיטיבית בין טיפוסים מוכללים ובין יחס ה-is-a
- המימוש של הכללות בג'אווה כולל מספר מוזרויות (ועוד לא דיברנו על כולן...)

- דיון מקיף (מעניין, וברור) בנושא ניתן למצוא בפרק 4 של:

  By David Flanagan Java in a Nutshell, 8th Edition