

# תוכנה 1 בשפת Java

## שיעור מספר 6: מקושרים



בית הספר למדעי המחשב  
אוניברסיטת תל אביב

# על סדר היום

---

- נתחיל בדוגמא נאיבית של מבנה מקושר
- נכליל את המבנה ע"י הכללת טיפוסים (Generics)
- נדון בייצוג הכרות אינטימית בשפת התכנות
- נדון בהפשטת מעבר סידרתי על נתונים והשלכותיו

# מבנים מקושרים

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

■ כדוגמא פשוטה ביותר, נגדיר מחלקה IntCell שעצמים בה מייצגים אברים ברשימות מקושרות של שלמים

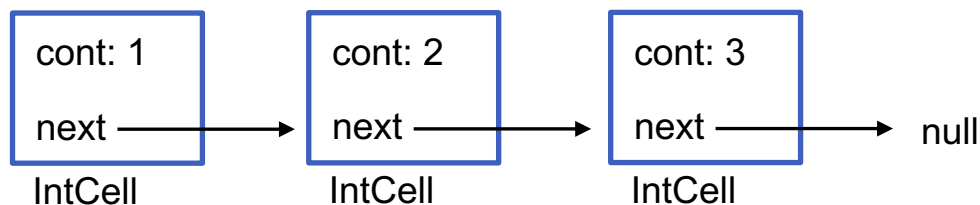
■ המחלקה מייצאת **בנאי** ליצירת עצם כאשר התוכן (שלם) והאבר הבא הם פרמטרים.

■ 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  
IntCell הוא מטיפוס

```
}
```

# 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 כגון `CharCell` או `StringCell`  
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



# מחלקות ושרותים מוכללים (גנריים)

■ החל מגירסא 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
- דוגמא 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;  
    }  
}
```

# 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.

■ לדוגמא: `Cell <String>`

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

(\* בהמשך הקורס נראה 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)



The image part with relationship ID rId3 was not found in the file.

# Boxing and Unboxing

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

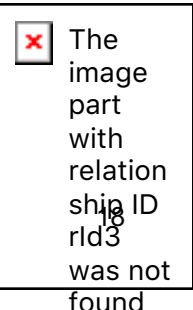
```
char pc = 'c';  
Character rc = new Character(pc);
```

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 `IntCell` שהשתמשה ב

```
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 ותא** כטיפוסים בשפה עם תכונות המתאימות לרמת ההפשטה שלהן
- נציג את המחלקה `MyList<T>` המייצגת רשימה

# קרוב ראשון ל- MyList<T>

```
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();  
    }  
}
```

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

# 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 `MyCell` להחליף את המימוש לרשימה דו-כיוונית

# MyList<T> - קרוב שני

```
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;
```

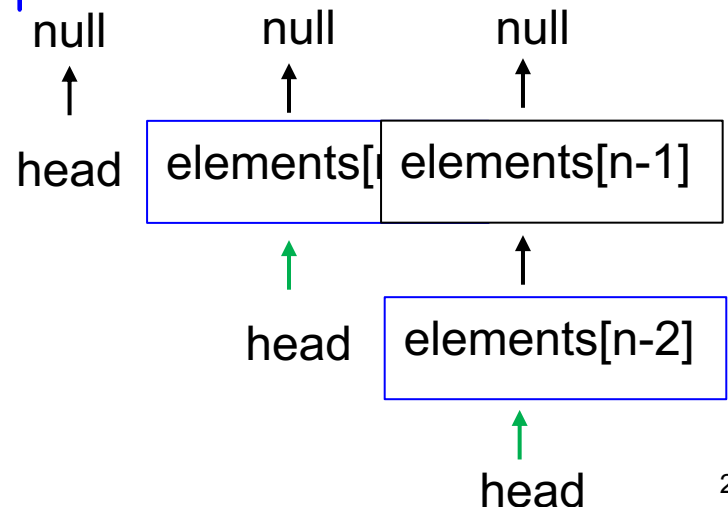
```
    }
```

```
    /** @pre !atEnd() */
```

```
    public void advance() {
```

```
        curr = curr.next();
```

```
    }
```

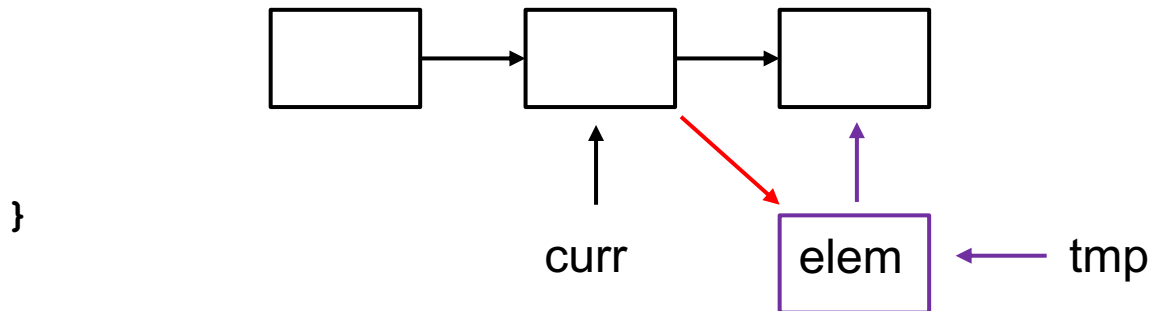


# המשך - MyList<T>

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

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

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





# 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  
y.cont() של הטיפוס T על toString

# MyList<T>

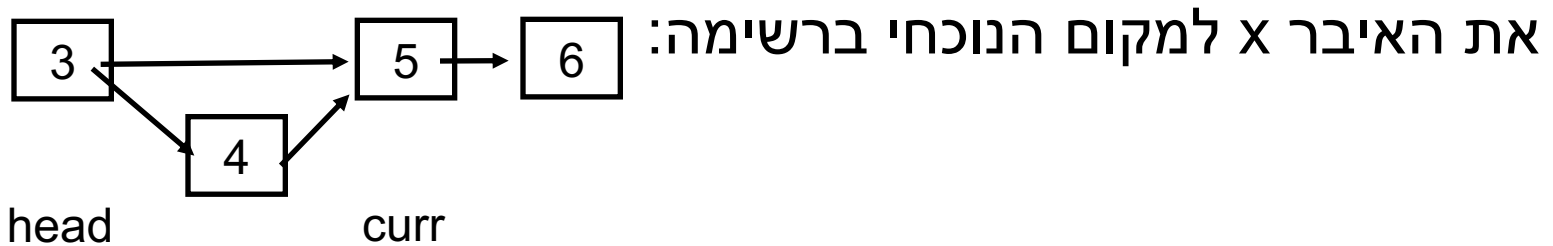
■ כעת לקוח הרשימה (MyList) אינו מודע לקיום מחלקת העזר  
:Cell<T>

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



# MyList<T>

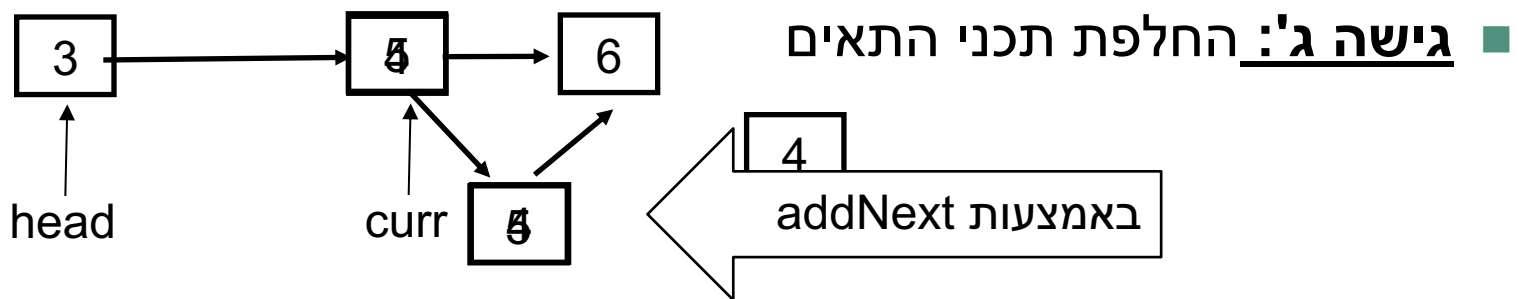
איך נממש את השרות `addHere(int x)` – שרות המוסיף

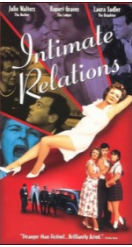


בשונה מהשרות `addNext()` אנו צריכים לשנות את ההצבעה לתא `curr`. לשם כך ניתן לנקוט כמה גישות:

■ גישה א': תחזוקה של `prev` נוסף על `curr`

■ גישה ב': נרוץ מתחילת הרשימה עד המקום אחד לפני הנוכחי (ע"י השוואת `next()` של כל תא ל `curr`)





# 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(elem);  
    curr.next().cont = curr.cont();  
    curr.cont = elem;  
}
```



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

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` | `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 כ**מחלקה 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)

# מחלקות פנימיות (מקוננות)

## Inner (Nested) Classes

# Inner Classes

■ מחלקה פנימית היא מחלקה שהוגדרה בתחום (Scope – בין המסולסליים) של מחלקה אחרת

■ דוגמא:

```
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 המסולסליים) של מחלקה אחרת

■ דוגמא:

```
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 כזה 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:

■ **Iterator** - **Collection**

■ **Brain** - **Body**

■ מבני נתונים המוגדרים ברקורסיה: **Cell** - **List**

# סוגי מחלקות פנימיות

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

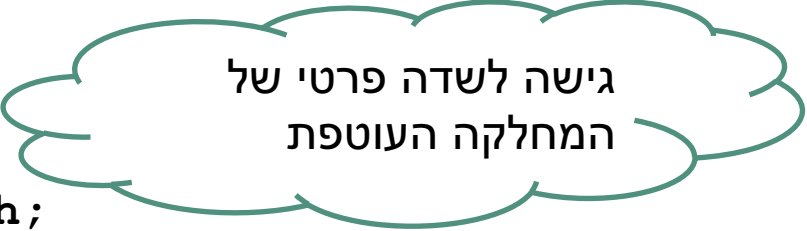
## ■ השלכות

- 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;  
    }  
}  
}
```



גישה לשדה פרטי של  
המחלקה העוטפת

# 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 כספרייה 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 (`effectively final`).

# 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 ארועים (כמו GUI) וילמדו בהמשך הקורס.



# 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 Cell | MyList

- כדי להסתיר מהלקוח של הרשימה את הייצוג הפנימי, וכדי 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 כגנרית. טיפוס התוכן של ה `Cell` נקבע על פי הפרמטר האקטואלי של עצם ה `MyList` המתאים.

- כלומר 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 מנתיב את פורמט הדפסה (כותרות, רווחים, שורות חדשות) ומגביל את הלקוח לפורמט זה. הלקוחה לא יכול לאסוף מידע זה בעצמה שכן הוא אפילו לא מכיר את המחלקה `Cell`