



תבנית עיצוב: המפעל

(factory design pattern)

# כמה מלים על תבניות עיצוב

- תבנית עיצוב היא פתרון מקובל לבעית תיכון נפוצה בתכנות מונחה עצמים.
- תבנית עיצוב מתארת כיצד לבנות מחלקות כדי לענות על הדרישה הנתונה.
- מספקת מבנה כללי שיש להשתמש בו כשמממשים חלק מתכנית.
- לא מתארת את המבנה של כל המערכת.
- לא מתארת אלגוריתמים ספציפיים.
- מתמקדת בקשר בין מחלקות.
- מתארת ניסיון מצטבר של מתכננים, שניתן ללמד ועוזר לתקשורת בין מהנדסי תוכנה.

# בנאים ומחלקת הלקוח

ניזכר בבנאי של המחלקה מלבן ובמתודה `bottomRight` ■

```
/** constructor using coordinates */  
public Rectangle(double x1, double y1, double x2, double y2) {  
    topRight = ???;  
    bottomLeft = ???;  
}
```

```
/** returns a point representing the bottom-right corner of the  
rectangle*/  
public IPoint bottomRight() {  
    return ???;  
}
```

כזכור, במקום סימני השאלה אמור להופיע בנאי של נקודה, ואולם למנשק `IPoint` אין בנאי, ואם נציין שם של בנאי של מחלקה קונקרטית אנו מפרים את חוסר התלות בין המלבן וקודקודיו ■

# בנאים ומחלקת הלקוח

■ **נסיון ראשון:** נגדיר במנשק IPoint את המתודה:  
IPoint createPoint() אשר תמומש בכל אחת  
מהמחלקות הממשות ליצור נקודה חדשה ולהחזיר  
אותה

■ **בעיה:** כדי להשתמש במתודה יש להפעיל אותה על  
עצמים שנוצרו כבר, בבנאי של Rectangle עוד לא  
נוצרה אף נקודה



# בנאים ומחלקת הלקוח

■ נסיון שני: נגדיר את המתודה כסטטית:

```
static IPoint createPoint()
```

■ בעיה: לא ניתן להגדיר במנשקים מתודות סטטיות



# שימוש במפעלים (factory design pattern)

- נגדיר מחלקה, שתכיל מתודה (אולי סטטית) שתפקידה יהיה להגדיר נקודות חדשות
- מחלקה כזו מכונה **מפעל (factory)**, והיא תהיה שדה במחלקה `Rectangle`
- לקוח טיפוסים של מלבן:

```
IPoint tr = new PolarPoint(3.0, (1.0/4.0) * Math.PI);
```

```
IPoint bl = new CartesianPoint(1.0, 1.0);
```

```
PointFactory factory = new PointFactory();
```

```
Rectangle rect = new Rectangle(bl, tr, factory);
```



```
public class PointFactory {
```

```
    public PointFactory(boolean usingCartesian, boolean usingPolar) {  
        this.usingCartesian = usingCartesian;  
        this.usingPolar = usingPolar;  
    }
```

```
    public PointFactory() {  
        this(false, false);  
    }
```

```
    public IPoint createPoint(double x, double y) {  
        if (usingCartesian && !usingPolar)  
            return new CartesianPoint(x, y);  
  
        if (usingPolar && !usingCartesian)  
            return new PolarPoint(Math.sqrt(x*x + y*y), Math.atan2(y, x));  
  
        return new SmartPoint(x, y);  
    }
```

```
    private boolean usingCartesian;  
    private boolean usingPolar;
```

```
}
```



```
public class Rectangle {
```

```
    private PointFactory factory;
```

```
    private IPoint topRight;
```

```
    private IPoint bottomLeft;
```

```
    /** constructor using points */
```

```
    public Rectangle(IPoint bottomLeft, IPoint topRight, PointFactory factory) {
```

```
        this.bottomLeft = bottomLeft;
```

```
        this.topRight = topRight;
```

```
        this.factory = factory;
```

```
    }
```

```
    /** constructor using coordinates */
```

```
    public Rectangle(double x1, double y1, double x2, double y2, PointFactory factory) {
```

```
        this.factory = factory;
```

```
        topRight = factory.createPoint(x1,y1);
```

```
        bottomLeft = factory.createPoint(x2,y2);
```

```
    }
```

כעת אין למחלקה Rectangle  
תלות במחלקת הנקודה כלל



# מדוע שימוש במפעלים עדיף?

- הרי עכשיו יש תלות בין המפעל ובין הנקודה, האם לא העברנו את הבעיה ממקום למקום?
- מחלקת המלבן היא מחלקה כללית, המיועדת לשימוש נרחב עם מגוון נקודות שכבר נכתבו ושטרם נכתבו
- מחלקת המלבן נוסף על היותה לקוח של המנשק IPoint משמשת גם ספק כלפי צד שלישי (שירצה ליצור מלבנים – למשל תוכנית גרפיקה)
- לקוחות המחלקה Rectangle הם אלו שצריכים להכיר את מגוון הנקודות הזמין לשימוש. מחלקת המפעל חוסכת מהם את ההתעסקות בפרטים אלה (פגיעה בהפשטה)
- שימוש במפעלים מדגיש את ההבדל בין הידע שיש לכותב ספרייה לעומת הידע שיש לכותב אפליקציה. זמינות המימושים (לדוגמא של טיפוס הנקודה) תהיה ידועה במלואה רק בזמן קונפיגורציה

# בנאים עם שם

## (named constructor idiom)

- נשתמש באותו הטריק של המפעל כדי "להעמיס בנאים" עם אותה חתימה
- מוטיבציה: המחלקה SmartPoint יודעת לטפל בצורה יעילה גם בייצוג קרטזי וגם בייצוג קוטבי. ואולם הבנאי שלה מקבל רק ייצוג קרטזי (כי לא ניתן להעמיס בנאים עם אותה חתימה)
- נוסף למחלקה את המתודות createPolar ו- createCartesian שיקבלו את שיעורי הנקודה המבוקשת בשני הייצוגים
- כדי להדגיש את הסימטריה של הייצוגים נהפוך את הבנאי לפרטי. כך לקוח מפוזר לא יוכל ליצור נקודה מבלי להיות מודע לייצוג שבו הוא משתמש

```
/** Default Constructor for private use */  
private SmartPoint(){  
}
```

```
public static SmartPoint createPolar(double r, double theta) {  
    SmartPoint result = new SmartPoint();  
    result.r = r;  
    result.theta = theta;  
    result.polar = true;  
    return result;  
}
```

```
public static SmartPoint createCartesian(double x, double y) {  
    SmartPoint result = new SmartPoint();  
    result.x = x;  
    result.y = y;  
    result.cartesian = true;  
    return result;  
}
```

# לסיכום

- מנשקים הם רכיב מפתח בעיצוב תוכנה
- הם אינם מייעלים בהכרח את קוד הספק
- מנשקים עשויים לתרום לחסכון בשכפול קוד לקוח
- פולימורפיזם מושג ב Java ע"י מנגנון ה dynamic dispatch – הפונקציה "המתאימה" תקרא בזמן ריצה
- ריבוי מנשקים מאפשר התמקדות בתכונות מסוימות של מחלקה (משקפיים)

תוכנה 1 בשפת Java  
שיעור מספר 6: מקושרים

**ליאור וולף**  
**ליאור שפירא**

בית הספר למדעי המחשב  
אוניברסיטת תל אביב

# על סדר היום

- נתחיל בדוגמא נאיבית של מבנה מקושר
- נכליל את המבנה ע"י הכללת טיפוסים
- 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).

■ 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 **שאליות** עבור התוכן והאבר הבא, ו**פקודות** לשינוי האבר הבא, ולהדפסת תוכן הרשימה מהאבר הנוכחי

■ השדות מוגדרים כפרטיים – מוסתרים מהלקוחות

■ המבנה `IntCell` אנלוגי למבנה `cons` משפת `Scheme`:

■ `cont ()` אנלוגי ל `car`

■ `next ()` אנלוגי ל `cdr`

# 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 שרות הראשי `main`.
- בהמשך הקורס נעסוק בנושא בדיקות (testing) אך כרגע נציין שעלינו 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

# 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)
- כרגע נציג רק את המקרה הפשוט. בהמשך נחזור לדון בנושא ביתר פירוט.
- 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 **Cell** 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 <Integer>` ■

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

# 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: `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 1.5 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();  
    }  
}
```

# עוד על שימוש בטיפוס גנרי

ניתן להגדיר משתנה (שדה, משתנה זמני, פרמטר) גם  
מהטיפוס `Cell <Cell <Integer>>`

מה מייצג הטיפוס הזה? ■

דוגמא של הצהרה עם אתחול: ■

```
Cell <Cell <Integer> > q =  
    new Cell <Cell <Integer>>  
        (new Cell<Integer> (8,null), null);
```



# מי אתה `Cell<T>` ?

■ האם `Cell<T>` באמת מייצג רשימה מקושרת?

■ בשפת Scheme התשובה היא כן. אולם 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);  
}
```

```
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>

■ 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>

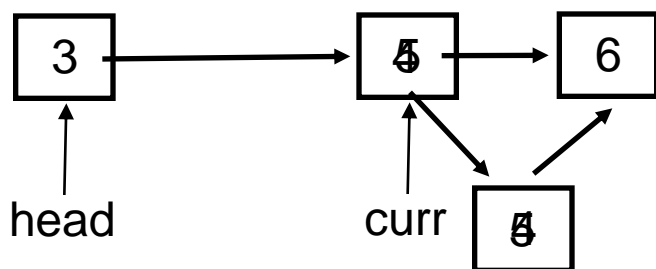
■ איך נממש את השרות `addHere(int x)` – שרות המוסיף



■ 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`)



■ גישה ג': החלפת תכני התאים

4

# 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;  
}
```

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 לאמר 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)

# 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
  - לעצם מטיפוס המחלקה הפנימית יש שדה הפנייה שמיוצר אוטומטית לעצם מהמחלקה העוטפת
  - כתוצאה מכך יש למחלה הפנימית גישה לשדות ולשרותים (אפילו פרטיים!) של המחלקה העוטפת ולהיפך
- 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`
- ניתן להגדיר מחלקה פנימית בתוך מתודה של המחלקה החיצונית
- ניתן להגדיר מחלקה פנימית אנונימית - הדבר שימושי במיוחד בספריות GUI. פרטים בהמשך הקורס

# 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
- 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 next() { return next; }  
        // ...  
    }  
}
```

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

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

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

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

```
    public void advance() { curr = curr.next(); }
```

```
    // ...
```

```
}
```

תוכנה 1 בשפת Java  
אוניברסיטת תל אביב

# דיון: `printList()`

■ `printList()` היא שרות גרוע

■ **בעיה:** השרות פונה למסך – זוהי החלטה שיש לשמור "לזמן קונפיגורציה". אולי הלקוח מעוניין להדפיס את המידע למקור אחר

■ **פתרון:** שימוש ב `toString` – שרות זה יחזיר את אברי הרשימה כמחרוזת והלקוח יעשה במחרוזת כרצונו

■ **בעיה:** השרות מכתוב את פורמט הדפסה (כותרות, רווחים, שורות חדשות) ומגביל את הלקוח לפורמט זה. הלקוח לא יכול לאסוף מידע זה בעצמו שכן הוא אפילו לא מכיר את המחלקה `Cell`

# Iterator Design Pattern

- נפתור בעיה זו ע"י שימוש בתבנית התיכון (תבנית עיצוב) Iterator
- Iterator אינו חלק משפת התכנות אלא הוא מייצג קונספט, רעיון, קלישאה תכנותית שמאפשרת לייצג את רעיון סריקת מבנה נתונים כללי
- בשפות תכנות מוכוונות עצמים (C++, Java, C#) ממומשים איטרטורים שימושיים כטיפוס בספריה הסטנדרטית

# איטרטור (סודר? אצן? סורק?)

- איטרטור הוא הפשטה של מעבר על מבנה נתונים כלשהו
- כדי לבצע פעולה ישירה על מבנה נתונים, יש לדעת כיצד הוא מיוצג
- גישה בעזרת איטרטור למבנה הנתונים מאפשרת למשתמש לסרוק מבנה נתונים ללא צורך להכיר את המבנה הפנימי שלו
- נדגים זאת על שני מבני נתונים המחזיקים תווים



# הדפסת מערך (אינדקסים)

```
char[] letters = {'a','b','c','d','e','f'};
```

```
void printLetters() {  
    System.out.print("Letters: ");
```

גישה בעזרת  
משתנה העזר  
לנתון עצמו

```
    for (int i=0 ; i < letters.length ; i++) {  
        System.out.print(letters[i] + " ");  
    }
```

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

```
}
```

הגדרת  
משתנה עזר  
ואתחולו

בדיקה:  
האם גלשנו

קידום משתנה  
העזר (מעבר לאיבר  
הבא)

# הדפסת רשימה מקושרת

```
public class MyList<T> {
```

```
    ...
```

```
    public void printList() {
```

```
        System.out.print("Letters : ");
```

```
        for (Cell<T> y = head; y != null ; y = y.getNext()) {  
            System.out.print(y.getCont() + " ");
```

הגדרת  
משתנה עזר  
ואתחולו

```
        System.out.print("\n");
```

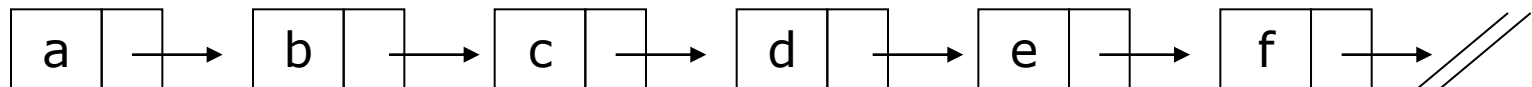
```
    }
```

```
}
```

בדיקה:  
האם גלשנו

גישה בעזרת  
משתנה העזר לנתון  
עצמו

קידום משתנה  
העזר (מעבר לאיבר  
הבא)



# הכרות אינטימית עם מבנה הנתונים

- 2 הדוגמאות הקודמות חושפות ידע מוקדם שיש לכותבת פונקצית ההדפסה על מבנה הנתונים:
  - היא יודעת איפה הוא מתחיל ואיפה הוא נגמר
  - היא מכירה את מבנה הטיפוס שבעזרתו ניתן לקבל את המידע השמור במצביע
  - היא יודעת איך לעבור מאיבר לאיבר שאחריו
- בדוגמת הרשימה המקושרת כותבת המחלקה `MyList` (הספקית) היא זו שכתבה את מתודת ההדפסה
- זה אינו מצב רצוי - זהו רק מקרה פרטי של פעולה אחת מני רבות של **שלקוחות** עשויים לרצות לבצע על מחלקה. על המחלקה לספק **כלים** ללקוחותיה לבצע פעולות כאלו בעצמם

# האיטרטור

■ איטרטור הוא בעצם **מנשק** (interface) המגדיר פעולות יסודיות שבעזרתן ניתן לבצע מגוון רחב של פעולות על אוספים

■ ב Java טיפוס יקרא Iterator אם ניתן לבצע עליו 4 פעולות:

■ בדיקה האם גלשנו (`hasNext ()`)

■ קידום (`next ()`)

■ גישה לנתון עצמו (`next ()`)

■ הסרה של נתון (`remove ()`)



# האיטרטור

■ כן, זה נורא! `next()` היא גם פקודה וגם שאילתה

■ ממש כשם שמימושים מסוימים של `pop()` על מחסנית גם מסירים את האיבר העליון וגם מחזירים אותו

■ בשפות אחרות (`C++` או `Eiffel`):

■ יש הפרדה בין קידום משתנה העזר והגישה לנתון

■ `remove()` אינה חלק משרותי איטרטור (וכך גם אנו סבורים)

# 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

■ מבנה הנתונים עצמו אחראי לספק ללקוח איטור תיקני (עצם ממחלקה שמממשת את ממשק `Iterator`) המאותחל לתחילת מבנה הנתונים

■ אם נרצה שהמחלקה `MyList` תספק ללקוחותיה את האפשרות לסרוק את כל האיברים בתוך העליון לכתוב `Iterator`

הגדרת  
משתנה עזר  
ואתחולו

בדיקה:  
האם גלשנו

# תקני MyListIterator

```
class MyListIterator<S> implements Iterator<S> {  
    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;  
    }  
  
    public void remove() {} // must be implemented  
  
    private Cell<S> curr;  
}
```

# MyList<T> מספקת איטרטור ללקוחותיה

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

- מחלקות המממשות את המתודה `iterator()` בעצם מממשות את הממשק `Iterable<T>` המכיל מתודה זו בלבד

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

- כעת 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);  
    }  
}
```

Autoboxing

What is the output for:

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

(שמרו לכן על הפרדה בין פקודות לשאיתות)

■ 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 הממשק `Iterable`

# for/in (foreach)

■ ראינו כי מערכים מתנהגים כטיפוס `:Iterable`

```
int[] arr = {6,5,4,3,2,1};  
for (int i : arr) {  
    System.out.println(i*i);  
}
```

■ שימוש נכון במבנה `for/in` מייתר רבים משימושי האיטרטור