# תוכנה 1 בשפת Java
# שיעור מספר 6: מקושרים

**סיון טולדו**
**אוהד ברזילי**

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).
- 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
- 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 `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

# 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 **<u>משתני טיפוס</u>** (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); // 3
```

# בחזרה לשימוש בטיפוס גנרי

- נראה מחלקה שמשתמשת ב **<T> Cell** , 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();
    }
}
```

# 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** ו**תא** 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

# קרוב שני - MyIntegerList

```
public class MyIntegerList {

    private Cell <Integer> head;
    private Cell <Integer> curr;

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

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

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

אנו דנים ב **MyIntegerList** ולא ב- **MyList<T>** מכיוון שאיננו רוצים להסביר בשלב זה את התחביר המסובך של טיפוס גנרי כללי, ואולם אין הבדל מהותי בין הדוגמאות

# MyIntegerList - המשך

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

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

```
    public void addNext(Integer elem) {
            Cell<Integer> temp = new Cell<Integer>(elem, curr.next());
            curr.setNext(temp);
    }

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

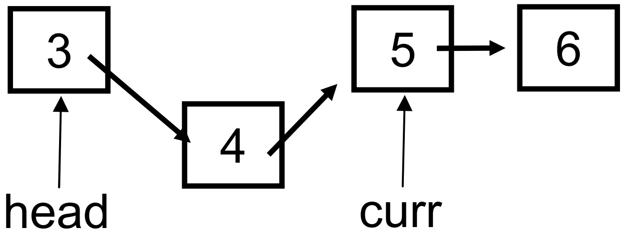
# MyIntegerList

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

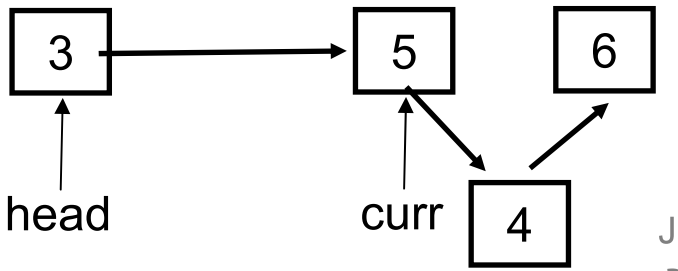
```
MyIntegerList l = new MyIntegerList(3,5);
l.printList();
l.advance();
l.addNext(4);
l.printList();
```

# MyIntegerList

- 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'

```
public void addHere(Integer elem) {
  addNext(elem);
  temp.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 **MyIntegerList** ולכן יש הצדקה למתן הרשאות גישה חריגות ל- **MyIntegerList** 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

- 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)

# 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;
    }
}
```

<u>שימו לב!</u>

Room אינה שדה של המחלקה House

# 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**

# Inner Classes

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

- **השלכות**
  - תחביר מיוחד לבנאי
  - לעצם מטיפוס המחלקה הפנימית יש שדה הפנייה שמיוצר אוטומטית לעצם מהמחלקה העוטפת
  - כתוצאה לכך יש למחלה הפנימית גישה לשרותים (אפילו פרטיים!) של המחלקה העוטפת

# Inner Classes

```
public class House {
    private String address;
    public class Room {
        // hidden reference to a House
        private double width;
        private double height;
        public String toString(){
            return  "Room inside: " +  address;
        }
    }
}
```

# Inner Classes

```
public class House {
    private String address;
    private double  height;
    public class Room {
        // hidden reference to a House
        private double height;
        public String toString(){
            return  "Room height: " + height
            + " House height: " + House.this.height;
        }
    }
}
```

Height of *House*

Height of *Room*

**Height of *Room***
**Same as *this.height***

# יצירת מופעים

- כאשר המחלקה החיצונית יוצרת מופע של עצם מטיפוס המחלקה הפנימית אזי העצם יווצר בהקשר של העצם היוצר

- כאשר עצם מטיפוס המחלקה הפנימית נוצר מחוץ למחלקה העוטפת, יש צורך בתחביר מיוחד

# יצירת מופע ע"י המחלקה החיצונית

```
public class House {
    private String address;
    public void test() {
        Room  r = new Room();
        System.out.println( r );
    }

    public class Room {
        ...
    }
}
```

# יצירת מופע שלא ע"י המחלקה החיצונית

```
public class Test {
   public static void main(String[] args){
         House  h = new House();
         House.Room  r =  h.new Room();
   }
}
```

***outerObject*.new *InnerClassName***

# Static Nested Classes

- 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`**

```java
public class House {
    private String address;
    public static class Room {
        public String toString(){
            return "Room " + address;
        }
    }
}
```

Error: this room is not related to any house

```java
public class Test {
    public static void main(String[] args){
        House.Room r = new House.Room();
        ...
```

Not related to any house

```
new OuterClassName.InnerClassName
```

# הגנה על מחלקות פנימיות סטאטיות

- אם המחלקה הפנימית אינה ציבורית (אינה מוגדרת public), הטיפוס שלה מוסתר, אבל עצמים מהמחלקה אינם מוסתרים אם יש התייחסות אליהם

```
public class Outer … {
   private static class Inner implement Inter {…}
   public static Inter getInner() {
      return new Inner ();
   }
    …

Inter i = new Outer.Inner();  //error
Inter i = Outer.getInner();   // ok
```

# מחלקות מקומיות -
# מחלוקת 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`** (מדוע?)

# מחלקות מקומיות

```
public class Test {
    ...
    public void test () {
            class Info {
                    private int x;
                    public Info(int x) {this.x=x;}
                    public String toString() {
                            return “** “ + x + “**” ;
                    }
            };
            Info  inf1 = new Info(0);
            System.out.println(inf1);
    }
}
```

# שימוש במשתנים מקומיים

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

# מחלקות אנונימיות

- בעזרת מחלקות פנימיות ניתן להגדיר מחלקות אנונימיות – מחלקות ללא שם
- מחלקות אנונימיות שימושיות מאוד במערכות מונחות ארועים (כגון 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 יהיה **Outer$Inner.class**
- אם המחלקה הפנימית אנונימית, שם המחלקה שיוצר הקומפיילר יהיה **Outer$1.class**

# 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 מכתיב 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#) מופיע ממומשים איטרטורים שימושיים **כטיפוס** בספריה הסטנדרטית

# איטרטור (סודר? אצן? סורק?)

- איטרטור הוא <u>הפשטה של מעבר</u> 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: ");

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

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

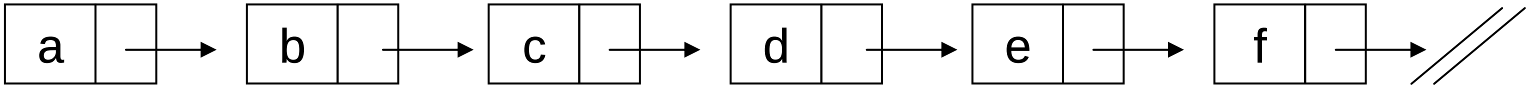
# הדפסת רשימה מקושרת

```
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() + " ");
        }

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



# 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 ש**לקוחות** 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()**) – אופציונלי

# 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());
}
```

- מבנה הנתונים עצמו אחראי לספק ללקוח איטרטור תיקני (עצם ממחלקה שמממשת 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> {
   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 המנשק **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 המנשק **`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