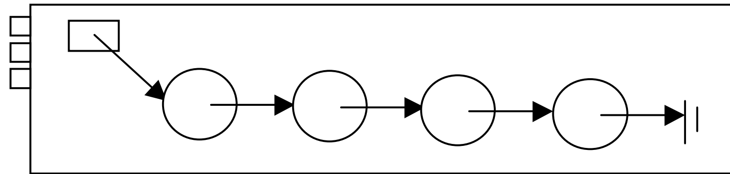
# איטרטורים

## מוטיבציה - התלבטות לגבי שירותים

בכדי להבין מאין עולה הצורך באיטרטורים, נחשוב על מבנה נתונים של רשימה משורשרת, חד כיוונית וננסה להגדיר לה, ממשק טוב. לשם הפשטות נניח שהיא מחזיקה int-ים.



כידוע, ממשק טוב לאובייקט הוא אחד הדברים החשובים ביותר בתכנות מונחה עצמים. אחד העקרונות המנחים בהגדרת ממשק, הוא השאיפה לממשק תמציתי וממצא (או מינימלי ושלם - minimal and complete). תמציתי - בגלל מודולריות: עם נרצה לשנות או להחליף את האובייקט, ככל שהממשק קטן יותר, נצטרך לדאוג להמשך תמיכה בפחות פעולות. ממצא - אנו מעונינים שהמתכנת-משתמש באובייקט שלנו יוכל לבצע עליו את כל הפעולות המתקבלות על הדעת ולא יהיה מוגבל בצורה שרירותית. שאיפה לממשק תמציתי וממצה מובילה לאוסף פונקציות שאינן חופפות וביחד מכסות את הפונקציונליות הדרושה. בהגדרת ממשק קיימים עד שיקולים רבים נוספים כמו, יעילות, נוחות שימוש ושמירה על אנקפסולציה. שיקולים אלה עומדים לפעמים בסתירה לרצון לשמור על מינימליות הממשק. במקרים רבים נדרשות פשרות.

הבה נחשוב מה סט השירותים אותו הגיוני לדרוש מרשימה משורשרת, ומה העלות של כל שירות בזמן ריצה:

- הוסף אבר להתחלה O(1).
- הסר אבר מההתחלה O(1).
- החזר את מספר האברים O(n) או O(1) עם תחזוקת שדה גודל.
- החזר את האבר ה - i-י. O(n).
- הוסף אבר לסוף הרשימה O(n) או O(1) ותחזוקה של מצביע לאבר האחרון.
- הסר את האבר האחרון ברשימה O(n).

כעת נניח שהמתכנת-משתמש ברשימה מעונין לכתוב קוד הבודק אם ערך מסוים נמצא ברשימה. הקוד שהוא יכתוב לשם כך, יכל להיראות כך:

```
boolean isIn(int data, List list) {
   for(int i=0; i<list.size(); ++i)
         if(list.elementAt(i)==data)
             return true;
   return false;
}
```

הקוד נראה בסדר, אך מה הסבוכיות שלו?
המימוש של הפונקציה ()elementAt ב - List, מן בסתם ניראה כך:

```
class List {
   …
  int elementAt(int i) {
         Node p;
         for(p = _head; i>0; --i)
             p = p._next;
         return p._data;
```

```
  }
    …
}
```

כפי שהקוד כתוב כרגע, בשביל לבדוק אם אבר נמצא ברשימה, יש צורך לעבור אבר, אבר ועבור כל אחד לרוץ מתחילת הרשימה. מספר הפעולות יהיה:

0+1+2+…+n = n*(n+1)/2 = n^2/2+n/2

כלומר, הסבוכיות היא O(n^2).

בניתוח שעשינו לא התייחסנו לזמן החישוב של size(). אפילו אם הסבוכיות של size היא ליניארית, אפשר לכתוב את isIn בצורה כזו שזמן החישוב שלה לא ישפיע בצורה משמעותית:

```
boolean isIn(int data, List list) {
   int size = list.size();
   for(int i=0; i<size; ++i)
         if(list.elementAt(i)==data)
             return true;
   return false;
}
```

כעת מדובר על תוספת ליניארית לזמן חישוב ריבועי. סוג כזה של ייעול קוד (אופטימיזציה) מבוצע לפעמים באופן אוטומטי, ע"י הקומפיילר.

הקוד שראינו, אם כן, בודק אם אבר נמצא ברשימה בזמן ריבועי. על פניו לא נראה שאלגוריתם כזה חייב להיות ריבועי, הרי בסך הכל אפשר לעבור על הרשימה בצורה סדרתית, ולבדוק אם הערך קיים. אפשר לכתוב קוד כזה:

```
boolean isIn(int data, List list) {
    for(Node p=list.getHead(); p != null; p=p._next)
          if(p._data == data)
              return true;
    return false;
}
```

כאשר getHead() ממומשת באופן הבא:

```
class List {
   …
   Node getHead() { return _head; }
   …
}
```

הסבוכיות של הקוד הזה היא אמנם ליניארית, אבל בשביל לכתוב אותו הפרנו עיקרון OO מקודש: האנקפסולציה. ברגע שנתנו למתכנת משתמש גישה ישירה למצביעי הקדקודים של הרשימה, הוא יוכל לפגוע בעקביות המבנה הפנימי של הרשימה והקוד שלו נעשה תלוי בפרטי המימוש שלה.

נראה, אם כן, שהגענו לסתירה בין יעילות לאנקפסולציה. אנו יודעים לכתוב קוד OO תקין אבל לא יעיל, וקוד יעיל אבל לא תקין. האם חייבים לוותר על אחד מהם? האם אי אפשר לכתוב קוד יעיל ותקין?

### ממשק שמן (fat interface)

אפשרות אחת היא להכניס את הפונקציה isIn לתוך List ולהכליל אותה ב"סל השירותים" שהיא מספקת:

```
class List {
```

```
   …
   public boolean isIn(int data) {
         for(Node p=_head; p != null; p=p._next)
            if(p._data == data)
                  return true;
         return false;
   }
   …
}
```

ואז המשתמש יוכל להפעיל אותה כך:

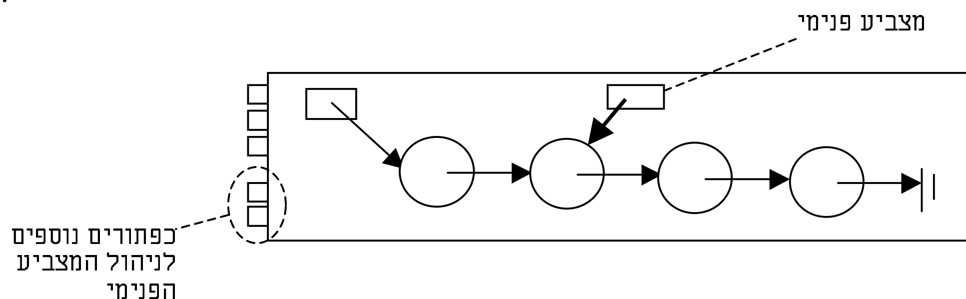
```
…
if(list.isIn(5)) …
```

פתרון זה הוא יעיל, אין בו פגיעה באנקפסולציה, אבל יש לו בעיה אחרת: פגיעה בממשק של List.
כזכור, בתחילת דברינו, נסינו להגדיר ל - List ממשק מינימאלי וממצה. האם isIn היא פונקציה הכרחית? אולי גם פונקציה הסופרת מספר מופעים של ערך ברשימה היא פונקציה שצריך להוסיף ל - List? אולי גם פונקציה המחשבת את סכום כל האברים של הרשימה?
קל לראות, שזו לא ארכיטקטורה מוצלחת. נהוג לכנות בעיה כזו בשם "fat interface", ממשק שמן, הוא ההפיך ממה שרצינו, ממשק תמציתי.

**פתרון המצביע הפנימי**

אנחנו עדיין נמצאים בהתלבטות לגבי הממשק של הרשימה. היינו מעונינים לתת סט שירותים מצומצם שמצד אחד מגן על המימוש הפנימי שלנו, ומצד שני מאפשר למשתמש לעשות את מה שהוא רוצה בצורה יעילה. לב הבעיה בדוגמה שלנו, היה הצורך של המשתמש במעבר סדרתי.
אפשר להוסיף לרשימה שלנו שירותים המאפשרים מאבר סדרתי עליה, באופן הבא:

```
class List {
   …
   private Node _position;
   public void reset() { _position = _head; }
   public void advance() { _position = _position._next; }
   public boolean atEnd() { return _position == null; }
   public int currentVal() { return _position._data; }
   …
}
```




ואז יוכל המשתמש ברשימה, לכתוב קוד כזה:

```
boolean isIn(int data, List list) {
         for(list.reset(); !list.atEnd(); list.advance())
            if(list.currentVal()==data)
                  return true;
         return false;
 }
```

למעשה הוספנו מצביע פנימי, המנוהל ע״י הרשימה ומופעל ע״י סט שירותים נוספים שנתנו למשתמש. כפי שאפשר לראות, סיבוכיות הקוד היא ליניארית והאנקפסולציה לא נפגעה. אמנם, הרחבנו את הממשק של הרשימה אבל נתנו אוסף קטן של שירותים נוספים שאפשר לממש אתו כל אלגוריתם הדורש מעבר סדרתי. המחיר היחידי ששילמנו הוא המקום בזיכרון הנדרש ל - _position בכל מקרה, גם אם לא משתמשים בו.

**איטרטורים**

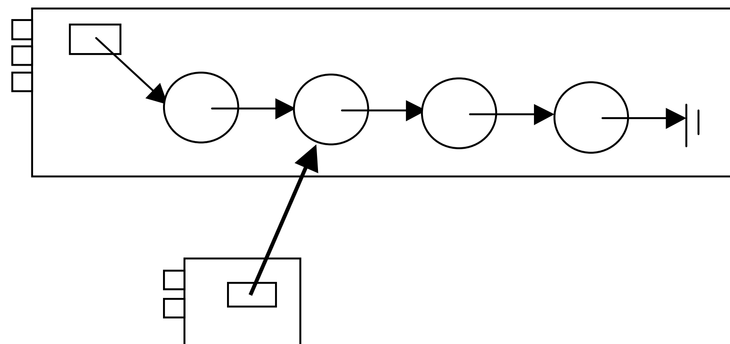
הבעיה המקורית שהוצגה אכן נפתרה ע״י הארכיטקטורה של המצביע הפנימי, אך הנה בעיה חדשה: המשתמש רוצה להדפיס את כל הזוגות הסדורים המורכבים מאלמנטים ברשימה. אם הייתה לו הפונקציה ()elementAt הוא היה כותב קוד כזה:

```
void printAllPairs(List list) {
   for(int i=0; i<list.size(); ++i) {
         int d1 = list.elementAt(i);
         for( int j=0; j<list.size(); ++j)
            System.out.println("<"+d1+","+list.elementAt(j)+">");
   }
}
```

כמובן שהקוד לא יעיל שוב, בגלל העבודה המיותרת שנעשית ע״י ()elementAt. כמו בבעיה הקודמת, אפשר היה לכתוב קוד יעיל ע״י שימוש ב - ()getHead ופגיעה באנקפסולציה.
בניגוד לדוגמה הקודמת, בבעיה זו לא נוכל להשתמש במצביע הפנימי מכיוון שאנו זקוקים כאן, למעבר סידרתי של שני מצביעים בו זמנית. בכל זמן נתון בשעת ביצוע האלגוריתם, כל מצביע אמור להחזיק מקום משלו. אפשר להרחיב את הפתרון של המצביע הפנימי ולספק שני מצביעים פנימיים ופונקציות המאפשרות להפעיל אותם. כמובן שיכל להתעורר צורך לריצה של יותר משני מצביעים בו זמנית. הרחבת הפתרון של המצביע הפנימי הופכת למסורבלת, המנגנון של ניהול המצביעים הפנימיים יהיה מבנה נתונים מסובך יותר מהרשימה המשורשרת עצמה.

בשלב זה נפנה לכיוון אחר, כיוון, שלפעמים יכל להיות מתגמל ביותר בתכנות מנחה עצמים. הבה ניתן לאובייקט מסוג אחר לתת את השירותים הנוספים של הרשימה, אותם שירותים שהרשימה מתקשה לתת בעצמה. האובייקט ישמור מיקום בתוך הרשימה, ויספק שירותים הכוללים התקדמות, החזרת הערך המוצבע ומענה על השאלה האם הוא נמצא בסוף הרשימה.

אותו אובייקט חיצוני המחזיק במיקום פנימי של מבנה הנתונים, ניקרא **איטרטור**. מבחינה ויזואלית, ניתן לראות זאת כך:



הקוד המתאים, יכל להראות כך:

```
public class Iterator {
   List.Node _position;
   public Iterator(List list) { _position = list._head;  }
```

```
   public int getVal() { return _position._data; }
   public void advance() { _position = _position._next; }
   public boolean atEnd() { return _position == null; }
}

public class List {
   static class Node {
         int _data;
         Node _next;
         Node(int data, Node next) { _data = data; _next = next; }
   }
   Node _head = null;
   public void add(int data) { _head = new Node(data,_head); }

   Iterator getIterator() { return new Iterator(this); }

}
```

ואז אפשר לכתוב את printAllPairs באופן הבא:

```
void printAllPairs(List list) {
      for(Iterator it1 = list.getIterator(); !it1.atEnd(); it1.advance())
         for(Iterator it2 = list.getIterator(); !it2.atEnd(); it2.advance())
            System.out.print("<"+it1.getVal()+","+it2.getVal()+"> ");
}
```

כפי שניתן לראות, הפתרון יעיל. אין כאן בזבוז של זמן על סריקות מיותרות. גם מבחינת אנקפסולציה הפתרון נראה טוב מכיוון שהמתכנת- משתמש במבנה הנתונים ובאיטרטור שלו, מפעיל רק פונקציות שאינן חודרניות. הפתרון גם אינו מוגבל לשתי סריקות המתבצעות במקביל. אפשר ליצור מספר כלשהו של אובייקטים מסוג איטרטור.

כאמור, מבחינת המתכנת-משתמש אין בפתרון שהוצג, פגיעה באנקפסולציה, אבל מה לגבי היחסים בין האיטרטור לרשימה? כפי שניתן לראות, האיטרטור מבצע ברשימה פעולות חודרניות המנצלות את הרשאות ה - package של _head ושל List.Node. אם היינו נותנים ל - _head ול-List.Node את הרשאת הגישה private, הקוד של Iterator לא היה עובר קומפילציה.

נשאלת השאלה האם העובדה ש - Iterator ניגש לקרביים של List, מהווה פגיעה באנקפסולציה? כזכור, המחלקה Iterator הוגדרה בכדי להרכיב את סט השירותים של List. הסיבה ששירותים אלו לא הוכנסו ל - List עצמה הם טכניים, ולא קשורים למהות של List. אם היינו יכולים להוסיף את השירותים הנוספים ל - List בלי לפגוע ביעילות או באנקפסולציה, היינו עושים זאת ולא מגדירים מחלקה חדשה.
מבחינת המשמעות, אם כן, Iterator ו- List אינם נפרדים, והם מהווים ביחד יחידה אחת של מבנה נתונים המאפשר מעבר סידרתי עליו. מכיוון ששתי המחלקות מתארות יחד, מהות אחת, הגיוני שתהיינה להן הרשאות חופשיות לגשת ל - member הפרטיים האחת של השניה. בקוד שהוצג ההרשאות ל - members היו הרשאות package, שאפשרו ל - Iterator לגשת לאינפורמציה לה היה זקוק. מתכנת משתמש שלא כותב באותה package, לא יוכל לגשת לאינפורמציה זו, אבל מתכנת משתמש הכותב באותה package יכול לגשת לאותם members והדבר מהווה פגיעה באנקפסולציה.
ב - C++ יש אפשרות שמחלקה אחת תצהיר על מחלקה אחרת כ"חברה" שלה (באמצעות המילה השמורה "friend"), ותאפשר לה לגשת לכל ה - members הפרטיים שלה. ב - Java אין את האפשרות הזאת, אך קיימת האפשרות לצור מחלקה פנימית עם הרשאות מיוחדות. הגדרת

Iterator כמחלקה פנימית של List מתאימה גם מבחינה קונספטואלית, באמת מדובר במחלקה שהצורך בה קיים מתוקף קיום List, ובאמת מתאים שהרשאות הגישה בין שתי המחלקות תהינה פתוחות לחלוטין. נשנה את הקוד הקודם כך ש - Iterator תהיה מחלקה פנימית של List:

```
public class List {
   static public class Iterator {
         Node _position;
         public Iterator(List list) { _position = list._head;  }
         public int getVal() { return _position._data; }
         public void advance() { _position = _position._next; }
         public boolean atEnd() { return _position == null; }
   }

   private static class Node {
         int _data;
         Node _next;
         Node(int data, Node next) { _data = data; _next = next; }
   }
   private Node _head = null;
   public void add(int data) { _head = new Node(data,_head); }

   Iterator getIterator() { return new Iterator(this); }
}
```

כעת ניתן לתת ל - List.Node ול - _head הרשאת private בלי שהדבר יפריע ל - Iterator לגשת אליהם.

כפי שנאמר בשעורים קודמים, ניתן להשתמש במחלקה פנימית לא סטטית ולחסוך את העברת המצביע לאובייקט היוצר ב - ctor של האובייקט הנוצר:

```
public class List {
   public class Iterator {
         Node _position = _head;          // same as: _position = List.this._head;
         … no ctor definition ..
   }
   …
   Iterator getIterator() { return new Iterator(); }
}
```

ב - Java נהוג לתת לאיטרטורים ממשק קצת שונה מממשק האיטרטור בדוגמאות שהבאנו.
הממשק שנהוג ב - Java לאיטרטור, כולל פונקציה המציינת אם קיימים עוד אברים לסריקה בשם hasNext() ופונקציה נוספת המחזירה את הערך הבא, ומקדמת את האיטרטור בשם next().
הפונקציה הראשונה היא פשוט השלילה של atEnd() והפונקציה השניה היא שילוב של getVal() ו - advance(). הקוד לפי הממשק הנהוג, יכל להיראות כך:

```
public class List {
   public class Iterator {
         Node _position = _head;
         public int next() {
            int ret = _position._data;
            _position = _position._next;
            return ret;
         }
         public boolean hasNext() { return _position != null; }
   }
   Iterator iterator() { return new Iterator(); } // the conventional function name
   …
}
```

ואז אפשר לממש את printAllPairs() כך:

```
static void printAllPairs(List list) {
         for(Iterator it1 = list.iterator(); it1.hasNext(); ) {
            int val1 = it1.next();
            for(Iterator it2 = list.iterator(); it2.hasNext(); )
                 System.out.print("<"+val1+","+it2.next()+"> ");
         }
}
```

שימו לב שבגלל ש - next() מאגדת בתוכה גם את החזרת הערך וגם את ההתקדמות, היינו חייבים לשמור את הערך שהוחזר ע"י it1 במשתנה זמני.
באופן כללי, הממשק הנהוג מחייב אותנו לקדם את האיטרטור כאשר אנו מעוניינים בערך.
לעיתים, קיבוץ שתי הפעולות בפעולה אחת עלול להפריע למשתמש לעשות את מה שהוא רוצה.
אם לדוגמה, האיטרטור מאפשר גם שינוי של הערך הנוכחי, והמשתמש רוצה לשנות את הערך רק אם הוא שווה למספר מסוים, אז בזמן שהמשתמש יבדוק מה הערך, האיטרטור יתקדם ולא יאפשר את השינוי המבוקש. לעומת זאת הממשק הנהוג עשוי להיות אינטואיטיבי ופשוט יותר לשימוש כאשר המשתמש רוצה פשוט לקבל את כל הערכים של מבנה הנתונים בצורה סדרתית.
אפשר כמובן להוסיף את getVal() לממשק הנהוג, ואז ליהנות, כביכול, משני העולמות.

הממשקים השונים של האיטרטור מדגימים שוב את ההתלבטות שבבחירת ממשק. הממשק הראשון שהבאנו היה בעל פונקציות אטומיות וזרות. הממשק השני (הנהוג) הכיל פחות פונקציות אך הן כבר לא היו אטומיות. הממשק השלישי שהצענו (הנהוג בתוספת getVal()) הכיל פונקציות שאינן זרות.
כפי שכבר אמרנו, לא תמיד יש תשובה ברורה, מהוא הממשק האופטימאלי. הבחירה היא tradeoff בין שיקולים שונים. העובדה שלא קיים ממשק טוב ביותר בצורה מובהקת לא צריכה להטעות לחשוב שכל ממשק הוא קביל. יש הרבה ממשקים שאפשר להעלות על הדעת שהם בעלי ממשק שמן שלא לצורך , שלא מאפשרים את כל עושר האפשרויות שסביר לתת למשתמש או שמחייבים את המשתמש לתכנות לא יעיל וכו'.

מכיוון שאיטרטור הוא מושג כללי המתאר יכולת לעבור על מבנה נתונים בצורה סדרתית, ניתן להגדיר interface המייצג איטרטור:

```
interface Iterator {
   public int next();
   public boolean hasNext();
}
```

המשתמש באיטרטור של List, איננו צריך לדעת את הסוג הספציפי של האיטרטור בו הוא משתמש, מספיק לו לדעת שמדובר באיטרטור. בשל כך ניתן לכתוב את הקוד בצורה כזו:

```
public class List {
   private class LIterator implements Iterator {
      … same code as in the previous iterator code …
   }
   Iterator iterator() { return new LIterator(); }
   …
}
```

בארכיטקטורה הזאת יש שני יתרונות: מבחינת אנקפסולציה אנחנו מגינים על הטיפוס הספציפי של האיטרטור וחושפים החוצה רק את הממשק הכללי של כל האיטרטורים. מבחינת גנריות ואבסטרקציה, אנו ממצים מכנה משותף של כל האיטרטורים ומאפשרים לכתוב קוד שהוא כללי (גנרי) מבחינת אופן הסריקה.

הקוד הבא ממחיש את האפשרות הזאת:

```
interface Iterator {
   public int next();
   public boolean hasNext();
}

public class List {
   private class LIterator implements Iterator {
         Node _position = _head;
         public int next() {
            int ret = _position._data;
            _position = _position._next;
            return ret;
         }
         public boolean hasNext() { return _position != null; }
   }
   Iterator iterator() { return new LIterator(); }

   private class SIterator implements Iterator {
         Node _position = _head;
         SIterator() { findNext(); }
         void findNext() {
            while(_position!=null && _position._data==0)
                  _position = _position._next;
         }
         public int next() {
            int ret = _position._data;
            _position = _position._next;
            findNext();
            return ret;
         }
         public boolean hasNext() { return _position != null; }
   }
   Iterator sIterator() { return new SIterator(); }
```

```
  private static class Node {
        int _data;
        Node _next;
        Node(int data, Node next) { _data = data; _next = next; }
  }
  private Node _head = null;
  public void add(int data) { _head = new Node(data,_head); }


  static int mulAll(Iterator it) {
        int mul = 1;
        while(it.hasNext())
           mul *= it.next();
        return mul;
  }

  static public void main(String[] args) {
        int[] array = {6,0,0,1,7,0,5};
        List list = new List();
        for(int i=0; i<array.length; ++i)
           list.add(array[i]);
        System.out.println(mulAll(list.iterator()));
        System.out.println(mulAll(list.sIterator()));
   }
}
```

פלט:

```
0
210
```

כפי שניתן לראות, האיטרטור הנוסף מדלג על כל המקומות שערכיהם שווים לאפס. האלגוריתם הכללי המחשב את מכפלת כל הערכים, מפיק תשובות שונות בהתאם לסוג האיטרטור שנשלח לו.

אורי מוסנזון GNU FDL FREE DOC LICENSE