שיכפול אובייקטים

הצורך בשכפול אובייקטים

לפעמים אנו מעונינים לבצע חישוב הדורש מאתנו לשנות נתונים אך איננו מעונינים לשנות את הערכים המקוריים. לדוגמה, אנו יכולים לבצע חישוב עם מערך של אנשים ובזמן החישוב למיין את המערך לפי תעודות הזהות שלהם, בכדי לזרז את החישוב. הסדר המקורי של המערך עשוי להיות חשוב לנו ולכן נעדיף אולי, לבצע את החישוב על מערך משוכפל.
פונקציות המקבלות פרמטרים פרימיטיביים, עובדות ממלא על שכפולים, אך פונקציות המקבלות refernces לאובייקטים עובדות למעשה על האובייקטים המקוריים.
סיבה נוספת לשכפול היא רצון לאכסן אובייקט במבנה נתונים כך שמבנה הנתונים יהיה בלתי תלוי באובייקטים חיצוניים. במקרה כזה נבחר להכניס למבנה הנתונים, העתק של האובייקט שבידינו.

שיכפול אובייקטים המאוכסנים במבני נתונים

כל מבני הנתונים הגנריים שראינו עד כה, אכסנו למעשה רק את המצביעים לאובייקטים שהם קיבלו. עובדה זו מאפשרת שינוי של תוכן מבנה הנתונים מבחוץ. לדוגמה:

```
class Check {
   static public void main(String[] args) {
         MyArray myArray = new MyArray();
         A a = new A();
         myArray.set(0,a);
         myArray.set(1,a);
         myArray.set(2,a);
         a.set(17);
         myArray.print();
   }
}

class MyArray {
   private Object[] _array = new Object[10];
   void set(int i, Object o) { _array[i] = o; }
   void print() {
         for(int i=0; i<10; ++i)
            System.out.print(_array[i]+"|");
   }
}

class A {
   int _a = 1;
   public void set(int a) {_a = a;}
   public String toString() {return ""+_a;}
}
```

פלט:

```
17|17|17|null|null|null|null|null|null|null|
```

כפי שניתן ליראות, השינוי שבצענו ב - a , לאחר הכנסתו, השפיע על מבנה הנתונים.

בכל מבני הנתונים הסטנדרטיים של Java, קיימת אותה תופעה:

```
import java.util.*;

class Check {
   static public void main(String[] args) {
         LinkedList list = new LinkedList();
         A a = new A();
         list.add(a);
         a.set(17);
         System.out.println(list.getFirst());
   }
```

```
}

class A {
   int _a = 1;
   public void set(int a) {_a = a;}
   public String toString() {return ""+_a;}
}
```

פלט: 17

ניתן ליראות בתופעה כפגיעה באנקפסולציה מכיוון שמתאפשר שינוי של תוכן מבנה הנתונים מבחוץ. מצד שני אין אפשרות לפגוע במבנה הפנימי של מיבנה הנתונים, רק בערכיו. בנוסף, הבעיה שתיארנו לא גורמת לתלות בקוד הפנימי של מיבנה הנתונים.
אם המשתמש במבנה הנתונים, מכיר את ההתנהגות שלו ויודע שהוא אינו משכפל אובייקטים אלא רק מחזיק מצביעים אליהם, הוא יוכל להשתמש בו בהתאם. הוא יוכל, לדוגמה, להימנע מלהשתמש באובייקט שניתן למבנה הנתונים:

```
list.add(new A(7));
```

או אפילו לנצל את חוסר השכפול:

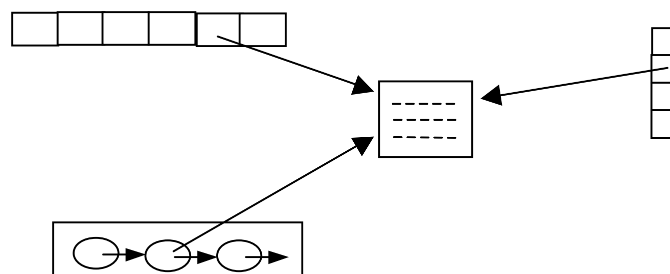
```
import java.util.*;

class Check {
   static public void main(String[] args) {
         LinkedList list = new LinkedList();
         ArrayList array = new ArrayList();
         A a = new A();
         list.add(a);
         array.add(a);
         ((A)list.getFirst()).set(17);
         System.out.println(array.get(0));
   }
}

class A {
   int _a = 1;
   public void set(int a) {_a = a;}
   public String toString() {return ""+_a;}
}
```

בדוגמה זו מוכנס אותו האובייקט מסוג A לשני מבני נתונים. שינויי של האובייקט מתוך אחד ממבני הנתונים משפיע על מבנה הנתונים השני. מתי התנהגות כזו עשויה להיות שימושית?
בתוכנה לניהול כוח אדם לדוגמה, התוכנה מחזיקה אובייקטים המתארים נתונים של אנשים. אובייקט של אדם מסוים יכול הימצא במספר מבני נתונים בו זמנית: מבנה הנתונים של העובדים הוותיקים, מבנה הנתונים של העובדים השיכים לועד העובדים וכו׳. ברגע שמקום המגורים של האדם משתנה, היינו רוצים שהפרטים יתעדכנו אוטומטית בכל מבני הנתונים. אם כל מבני הנתונים מחזיקים רק מצביע לאותו אובייקט, שינוי האובייקט ישפיע על כולם באופן אוטומטי.

האיור הבא ממחיש את המצב בו אובייקט אחד מוצבע ע״י מספר מבני נתונים:



למרות כל הדברים שנאמרו בזכות מבני נתונים לא משכפלים, לעתים אנו מעונינים במבנה נתונים בלתי תלוי בעולם החיצוני. מבנה נתונים שיוכל לשנות את ערכיו הפנימיים ללא חשש מהשפעה על חלקים אחרים של התוכנית וללא חשש משינוי הערכים שברשותו ע"י חלקים אחרים בתוכנית.

ניתן לשנות את מבנה הנתונים MyArray ולהפוך אותו למבנה נתונים משכפל:

```
class Check {
   static public void main(String[] args) {
          MyArray myArray = new MyArray();
          A a = new A();
          myArray.set(0,a);
          myArray.set(1,a);
          myArray.set(2,a);
          a.set(17);
          myArray.print();
   }
}

class MyArray {
   private Object[] _array = new Object[10];
   void set(int i, Copiable o) { _array[i] = o.getCopy(); }
   void print() {
          for(int i=0; i<10; ++i)
             System.out.print(_array[i]+"|");
   }
}

interface Copiable {
   public Object getCopy();
}

class A implements Copiable {
   int _a = 1;
   public void set(int a) {_a = a;}
   public String toString() {return ""+_a;}
   public Object getCopy() {
          A a = new A();
          a._a = _a;
          return a;
   }
}
```

פלט:

```
1|1|1|null|null|null|null|null|null|null|
```

כעת נוסיף גם למבנה הנתונים MyArray את האפשרות לייצר שיכפול. כאובייקט הניתן לשכפול, אפשר יהיה להכניס אותו למבנה הנתונים MyArray.

```
Class Check {
   static public void main(String[] args) {
          MyArray myArray = new MyArray();
          myArray.set(0,new A(1));
          myArray.set(1,new A(2));
          myArray.set(2,myArray);
          System.out.println(myArray);
          myArray.set(1,new A(17));
          System.out.println(myArray);
```

```
    }
}

class MyArray implements Copiable {
    private Object[] _array = new Object[5];
    public void set(int i, Copiable o) { _array[i] = o.getCopy(); }
    public Object getCopy() {
            MyArray myArray = new MyArray();
            for(int i=0; i<5; ++i)
                if(_array[i] != null)
                        myArray._array[i] = ((Copiable)_array[i]).getCopy();
            return myArray;
    }
    public String toString() {
            String s = "[";
            for(int i=0; i<5; ++i)
                s+=_array[i]+"|";
            return s+"]";
    }
}

interface Copiable {
    public Object getCopy();
}

class A implements Copiable {
    private int _a = 1;
    public A(int a) {_a = a;}
    public void set(int a) {_a = a;}
    public String toString() {return ""+_a;}
    public Object getCopy() { return new A(_a); }
}
```

פלט:

```
[1|2|[1|2|null|null|null|]|null|null|]
[1|17|[1|2|null|null|null|]|null|null|]
```

התבוננו בשורת הקוד הבאה:

```
myArray.set(2,myArray);
```

ונסו לחשוב למה הייתה גורמת אילו MyArray לא היה מבנה נתונים משכפל.

ניתן דוגמה נוספת למבנה נתונים הניתן לשכפול, הפעם עץ בינארי ממוין:

```
class Tree implements Copiable {
    static private class Node {
            int _data;
            Node _ls,_rs;
            Node(int data) { _data = data; }
    }
    private Node _root;

    public void insert(int data) {
            if(_root == null) {
                _root = new Node(data);
                return;
            }
            Node p = _root;
            while(true) {
```

```
            if(data < p._data)
                    if(p._ls != null)
                        p = p._ls;
                    else {
                        p._ls = new Node(data);
                        break;
                    }
            else
                    if(p._rs != null)
                        p = p._rs;
                    else {
                        p._rs = new Node(data);
                        break;
                    }
        }
    }

    public String toString() {
            return toString(_root);
    }

    static private String toString(Node p) {
            if(p == null)
                return "n";
            return "(" + toString(p._ls) + "," + p._data + "," +
                toString(p._rs) + ")";
    }

    public Object getCopy() {
            Tree tree = new Tree();
       tree._root = getCopy(_root);
            return tree;
    }

    static private Node getCopy(Node p) {
            if(p==null)
                return null;
            Node p1 = new Node(p._data);
            p1._ls = getCopy(p._ls);
            p1._rs = getCopy(p._rs);
            return p1;
    }

    static public void main(String[] args) {
            int[] array = {7,2,4,3,1,6,5};
            Tree tree = new Tree();
            for(int i=0; i<array.length; ++i)
                tree.insert(array[i]);
            Tree tree1 = (Tree)tree.getCopy();
            tree.insert(1777);
            System.out.println(tree);
            System.out.println(tree1);
    }
}

interface Copiable {
    public Object getCopy();
}
```

פלט:

```
(((n,1,n),2,((n,3,n),4,((n,5,n),6,n))),7,(n,1777,n))
(((n,1,n),2,((n,3,n),4,((n,5,n),6,n))),7,n)
```

מנגנון ה - clone ב - Java

ב - Java קיים מנגנון פנימי הדואג להעתקה של אובייקטים. כאשר רוצים לשכפל אובייקט קיים, מפעילים את הפונקציה clone שלו ומקבלים העתק. לדוגמה:

```
import java.util.*;

class Check {
   static public void main(String[] args) {
         ArrayList a1 = new ArrayList();
         for(int i=0; i<7; ++i)
            a1.add(new A());
         ArrayList a2 = (ArrayList)a1.clone();
         ((A)a1.get(3))._a = 1000;
         a1.remove(0);
         System.out.println(a1+"\n"+a2);
   }
}

class A {
   public int _a = 1;
   public String toString() {return Integer.toString(_a);}
}
```

פלט:

```
[1, 1, 1000, 1, 1, 1]
[1, 1, 1, 1000, 1, 1, 1]
```

כפי שניתן לראות, הפונקציה clone של ArrayList מחזירה העתק של המערך אבל לא של הערכים השמורים בו. העתק כזה נקרא העתק רדוד - shallow copy בניגוד להעתק המלא שביצענו ב - MyArray שנקרא "העתק עומק" - deep copy.
ישנן מספר סיבות מדוע המערך הגמיש של Java לא מבצע העתקת עומק. סיבה אחת היא שהמערך לא יכול להניח כי האובייקטים שהוא מחזיק, ניתנים להעתקה. למעשה, רוב האובייקטים בספרייה הסטנדרטית של Java, אינם ניתנים להעתקה.
ישנה סיבה נוספת הקשורה לאופן המימוש של clone אבל אותה ניראה בהמשך.

הפונקציה ()clone ממומשת במחלקה Object ולכן נורשת ע"י כל מחלקה ב - Java. בשביל למנוע אפשרות שכפול מכל מחלקה שלא מממשת שיכפול באופן מפורש הרשאת הגישה של ()Object.clone היא protected. התבוננו בקוד הבא:

```
class Check {
   static public void main(String[] args) {
         A a = new A();
         // a.clone(); // c.error: Object.clone is protected
   }
}

class A{
   void foo() { clone(); }
}
```

כפי שניתן לראות, לא ניתן לקרוא ל clone מחוץ למחלקה מכיוון שהיא protected אבל ניתן לקרוא לה מתוך פונקציה של A מכיוון ש - A יורשת את Object (למעשה קוד זה לא יעבור קומפילציה בסיבה הקשורה ל - exception, עניין שיובהר בהמשך).

בכדי שאובייקט יהיה ניתן לשכפול, הוא צריך לממש את הממשק Cloneable. הממשק Cloneable הוא ממשק ריק, שנועד רק לציין את העובדה שהאובייקט ניתן לשכפול. הדבר הבא שיש לעשות הוא לממש פונקציה **ציבורית** בשם clone. כפי שכבר נאמר, Object.clone() היא פונקציה protected ולכן ניתן לבצע לה override ע"י פונקציה עם הרשאת גישה מתירנית יותר. אם מתכנת ינסה לשכפל אובייקט שלא בצע overriding ציבורי לפונקציה, הוא יקבל אינדיקציה בזמן קומפילציה שניסה לשכפל אובייקט שלא אמור להיות משוכפל.
המימוש של clone אמור תמיד לכלול קריאה ל - clone של ה - super. באופן זה אמורה להיקרא בסופו של דבר, הפונקציה Object.clone(). הפונקציה Object.clone() מבצעת שתי פעולות:
האחת היא לבדוק שהמחלקה הספציפית של האובייקט אכן מממשת את Cloneable והשניה היא לבצע את ההעתקה של האובייקט. אם הפונקציה מגלה שהמחלקה הספציפית לא מממשת cloneable, היא זורקת Exception מסוג CloneNotSupportedException. ההעתקה שהפונקציה מממשת היא העתקה פשוטה של המקום בזיכרון. העתקה כזאת נקראת bitwise copy. יש לשים לב שהעתקה כזאת מעתיקה משתנים פרימיטיביים אך לא מעתיקה אובייקטים (היא מעתיקה רק את המצביעים אליהם). ההעתקה של Object.clone() היא בעצם סוג של shallow copy. אם מעוניינים בהעתקת עומק, יש לכתוב לך קוד מפורש נוסף.
ניתן דוגמה בסיסית למימוש מחלקה הניתנת להעתקה:

```
class A implements Cloneable{
   public int _a;
   public A(int a) {_a = a; }
   public Object clone() {
          Object o=null;
          try {
             o = super.clone();
          } catch (CloneNotSupportedException e) { }
          return o;
   }
}
```

ניתן להשתמש בה באופן הבא:

```
class Check {
  public static void main(String[] args) {
     A
            a = new A(7),
            a1 = (A)a.clone();
     a._a++;
     System.out.println(a._a+ " " + a1._a);
  }
}
```

פלט: 8 7

הקוד הבא מדגים העברה של אובייקט מסוג A כביכול - by value:

```
class Check {
   static private void foo(A a) {
          a = (A)a.clone();
          a._a = 177;
          System.out.println(a._a);
   }
```

```
    public static void main(String[] args) {
            A a = new A(7);
            foo(a);
            System.out.println(a._a);
    }
}
```

פלט:

177
7

הקוד הבא מדגים את העובדה שכל שיכפול נמצא במקום נפרד בזיכרון:

```
public static void main(String[] args) {
            A a1,a2,a3,a4;
            a1 = a2 = new A(17);
            a3 = (A)a1.clone();
            a4 = (A)a2.clone();
            System.out.println((a1==a2) + " " + (a2==a3) + " " + (a3==a4));
    }
```

פלט:
true false false

הקוד הבא מדגים שההעתקה המבוצעת ע"י Object.clone היא בעצם סוג של shallow copy:

```
class Try {
    static public void main(String[] args) {
            A a = new A(new B()), a1 = (A)a.clone();
            a._b._b = 8;
            System.out.println(a1);
    }
}

class A implements Cloneable{
    public B _b;
    public A(B b) {_b = b;}
    public String toString() { return _b + ""; }
    public Object clone() {
            Object o=null;
            try {
                o = super.clone();
            } catch (CloneNotSupportedException e) {}
            return o;
    }
}

class B {
    public int _b=1;
    public String toString() { return ""+_b; }
}
```

פלט: 8.

בכדי שהמחקה A תשוכפל בהעתקת עומק יש להוסיף ל clone() את הקוד הבא:

```
public Object clone() {
            Object o=null;
            try {
                o = super.clone();
            } catch (CloneNotSupportedException e) {}
            int tmp = _b._b;
```

```
        _b = new B(); // could use _b.clone() if B was colneable
        _b._b = tmp;
        return o;
    }
```

כאשר יורשים cloneable הוא אוטומטית cloneable. אם התוספת היחידה היא משתנים פרימיטיביים, אין צורך להוסיף שום קוד לשכפול:

```
class Try {
    static public void main(String[] args) {
        B b = new B(1,2), b1 = (B)b.clone();
        b._a +=5;
        System.out.println("b: "+b+"\nb1: "+b1);
    }
}

class A implements Cloneable{
    public int _a;
    public A(int a) {_a = a; }
    public Object clone() {
        Object o=null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {}
        return o;
    }
}

class B extends A {
    // no clone code is needed
    private int _b;
    public B(int a,int b) {
        super(a);
        _b = b;
    }
    public String toString() {
        return _a + " " + _b ;
    }
}
```

פלט:

```
b: 6 2
b1: 1 2
```

הדוגמה הזאת מראה את היתרון הגדול של שימוש במנגנון השכפול המובנה של Java לשכפול אובייקטים. אם היינו ממשים דוגמה זו ללא המנגנון, כפי שמימשנו את getCopy, היינו צריכים לממש גם במחלקה B, פונקציה הדואגת להעתקה. אותה פונקציה שהיינו כותבים - B.getCopy() הייתה מכילה שוב את הקוד שהופיע כבר ב - A.getCopy(). בגלל ש Object.clone() מממשת bitwise copy של רצף הזיכרון **בגודל האובייקט הספציפי**, נחסך מאתנו אותו שכפול קוד.

Copy ctor

יש אפשרות לכתוב את הקוד של הדוגמה האחרונה ללא שימוש ב - clone ושעדיין יהיה נקי משכפולי קוד. הדרך לעשות זאת היא להשתמש במנגנון בניית אובייקט נורש ולממש ctor נוסף היודע לבנות שיכפול של אובייקט קיים:

```
class Check {
    static public void main(String[] args) {
        B b = new B(1,2), b1 = (B)b.getCopy();
```

```
        b._a +=5;
        System.out.println("b: "+b+"\nb1: "+b1);
    }
}

class A implements Copiable{
    public int _a;
    public A(int a) { _a = a; }
    public A(A other) {_a = other._a;}
    public Object getCopy() { return new A(this); }
}

class B extends A {
    private int _b;
    public B(int a,int b) {
        super(a);
        _b = b;
    }
    public B(B other) {
        super(other);
        _b = other._b;
    }
    public Object getCopy() { return new B(this); }
    public String toString() {
        return _a + " " + _b ;
    }
}
```

כפי שניתן לראות, הפונקציה getCopy() פשוט יוצרת אובייקט חדש תוך שימוש ב - ctor משכפל. אותו ctor כולל רק את קוד הבניה של החלק הספציפי של היורש ומשאיר את בניית החלק הנורש בידי ה - ctor של המחלקה המורישה. ל - ctor המתאר איך לבנות אובייקט חדש כהעתק של אובייקט קיים מאותו סוג, קוראים copy ctor.

הסרבול של clone ב - Java

כפי שכבר ראינו, מנגנון ה - clone ב - Java כולל שני סוגים של אבטחות נגד שיכפול של אובייקט שלא יועד לכך. אבטחה אחת מתבטאת בזמן קומפילציה - אם המחלקה לא מממשת clone כ - public אז כל ניסיון חיצוני, לבקש מהאובייקט להשתכפל לא יעבור קומפילציה. האבטחה השניה מתבטאת בזמן ריצה: אם המחלקה הספציפית של האובייקט לא מממשת את ממשק התג - cloneable, ייזרק בזמן ריצה exception ע״י Object.clone(). יש לשים לב שבעד שהאבטחה שבזמן ריצה, גורמת לסרבול קטן בקוד השכפול, האבטחה שבזמן קומפילציה ממש מגבילה את אפשריות השימוש ב - clone. ההגבלה נובעת מהעובדה שאי אפשר לשכפל אובייקט דרך התייחסות אליו כ - Obect:

```
class MyArray implements Cloneable {
   Object[] _array = new Object[10];
   public Object clone() {
         MyArray ret = new MyArray();
         for(int i=0; i<10; ++i)
            ret._array[i] = _array[i].clone(); // c.error: Object.clone() is protected
         return ret;
   }
}
```

אם Object.clone() הייתה ציבורית אז אפשרות כזאת לא הייתה נמנעת.
טבעי לתהות מדוע מעצבי השפה Java הגנו על clone בצורה מגבילה כל כך. האמת היא שבמקור, כאשר Java היית עד באורינטציה של שפה משובצת מכשירים, Object.clone() הייתה פונקציה ציבורית. רק מאוחר יותר, כאשר Java הפכה לשפה המיועדת לשימוש נרחב ברשת ה - internet, התגלו בעיות אבטחה, ומתכנני השפה נאלצו להוסיף אבטחות למנגנון ה - clone(). ברור שהוספת האבטחות פגמה במנגנון ובנוחות השימוש בו.