

חלק 8

המחלקה פּיִּישׁוֹת

בפני עצמה

חיסכון ביצירת עצמים

- ננסה למחזר עצמים מסוג Version שאינם נחוצים יותר; כאשר לא צריך יותר עצם מסוים, נחזיר אותו למאגר של עצמים חופשיים
- כאשר צריך עצם חדש מהמחלקה, אם יש במאגר עצם קיים שאינו בשימוש, נשתמש בו, אחרת ניצור עצם חדש
- זה כמובן דורש שלקוחות שצריכים עצם כזה ימנעו מקריאה לבנאי, כי זה תמיד ייצור עצם חדש

```
class Version {  
    private static Version free_list;  
    private          Version() {}  
    public static Version alloc() {...}  
    public          Version free() {...}
```

חיסכון ביצירת עצמים (המשך)

```
class Version {  
    private static Version free_list;  
    private          Version () {}  
    public static Version alloc() {...}  
    public          void    free() {...}  
  
    String value;  
    Version previous;  
}
```

- לקוחות לא יכולים לבנות עצמים ישירות
- הפרוצדורה alloc כן יכולה לבנות עצמים

הקצאה ושחרור

```
public static Version alloc() {
    if (free_list == null)
        return new Version();
    else {
        Version v = free_list;
        free_list = v.previous;
        return v; } }

public void Version free() {
    this.value      = null;
    this.previous   = free_list;
    free_list       = this; }
```

שדות מחלקה ושירותי מחלקה

- השדה `free_list` משותף לכל העצמים מהמחלקה; שדה מוכרז כשדה מחלקה (להבדיל משדה מופע) בעזרת מילת המפתח `static`
- בשירות של המחלקה, השם `previous` קשור לשדה של העצם שהפעיל את השירות (שדה מופע)
- אבל השם `free_list` קשור לשדה משותף לכל העצמים מהמחלקה (שדה מחלקה, `class field`)
- `alloc` היא פרוצדורה או שירות מחלקה (`class method`); היא מופעלת ישירות על ידי `Version.alloc()` ולא על עצם שהוא מופע של המחלקה; יש לה גישה לשדות המחלקה, והיא יכולה להשתמש במחלקה כלקוח, אבל כלקוח יש לה גישה גם לשדות ושירותים מוגנים (`private` וכדומה)

למה שדות מחלקה?

- כי הם מהווים שמות גלובליים (כלומר התייחסות שתמיד אפשר למצוא, לעומת עצמים רגילים שצריך לקבל מאיזשהו עצם אחר התייחסות אליהם)
- כי הם יחידים (כלומר יש בדיוק `Version.free_list` אחד בתוכנית)
- זה בדיוק מה שרצינו: להיות מסוגלים למצוא את מאגר העצמים הפנויים, גם איננו מכירים אף עצם מהמחלקה, ושיהיה רק מאגר פנויים אחד

הערות לגבי הקצאת זיכרון

- בג'אווה הזיכרון מנוהל אוטומטית; כל עוד יש התייחסות לעצם, הזיכרון שלו לא יוקצה למטרה אחרת, ואם אין התייחסות לעצם, הזיכרון שלו יוחזר למערכת (אולי לא מייד)

- למרות זאת, הדוגמה מראה שאפשר להגדיר מחלקות שמממשות ניהול זיכרון מפורש, ולכן גם אפשר ליצור תוכניות עם פגמים בניהול הזיכרון, כמו בשפות ללא ניהול אוטומטי

```
Version v1 = Version.alloc();
```

... *we use the object that v1 refers to*

```
v1.free();
```

```
Version v2 = Version.alloc();
```

```
v1.value = "something"; an error!
```

- בדרך כלל הפגם יותר קשה לגילוי מאשר בדוגמה

הקצאת זיכרון (ומשאבים אחרים)

- דליפת זיכרון (memory leak) הוא פגם אחר בניהול זיכרון מפורש: עצם מוקצה אבל לא משוחרר מפורשות (האם דליפת זיכרון היא פגם בדוגמה שלנו?)

```
java.io.FileInputStream is
```

```
= new FileInputStream("C:\grades.dat");
```

```
... we the input stream
```

```
is.close(); we release the object; shouldn't use it later
```

- מערכת ההפעלה מאפשרת לפעמים לפתוח רק מספר מוגבל של קבצים, ולכן קובץ פתוח הוא משאב מוגבל; חשוב לשחרר את המשאב ברגע שלא צריכים אותו
- אי שחרור בזמן הוא פגם בתוכנית, ושימוש בקובץ ששוחרר גם הוא פגם

עוד שימושים לשדות מחלקה

- יש עוד תבניות תיכון (design patterns) שמתמשים בשדות מחלקה
- תבנית היחיד מבטיחה שיש רק עצם אחד ממחלקה מסוימת (singleton pattern)
- תבנית משקל נוצה (flyweight pattern) יוצרת מאגר של עצמים מקובעים ודואגת שבכל רגע נתון יהיה לכל היותר עצם אחד עם מצב מופשט נתון; זה חוסך זיכרון ומאפשר לבצע השוואות עם == במקום equals
- השירות `java.lang.String.intern()` מתחזק תבנית כזו: הוא מחזיר התייחסות לעותק קנוני של מחרוזת

קבועים

- שדות מחלקה משמשים גם לציון קבועים בתוכנית
- מוסכמה סגנונית: בדרך כלל שמות שכוללים רק אותיות גדולות

```
class Paragraph {  
    public final static int DIR_LTR      =0x01;  
    public final static int DIR RTL     =0x02;  
    public final static int ALIGN_LEFT  =0x04;  
    public final static int ALIGN_RIGHT=0x08;  
    public Paragraph (int style) {...}  
    ...  
}
```

שימוש בקבועים

```
class Paragraph {  
    public final static int DIR_LTR      =0x01;  
    ...  
    public Paragraph (int style) {...}  
    ...  
}  
  
...  
Paragraph p = new Paragraph(  
    Paragraph.DIR_RTL | Paragraph.ALIGN_RIGHT  
);
```

המשתמר של שדות המחלקה

- (נניח את ההנחה הסבירה ששדות המחלקה מוגנים מגישה ישירה על ידי לקוחות של המחלקה ולכן אינם מופיעים בחוזה)
- מי משתמש בשדות המחלקה?
- שירותי מחלקה ושירותי מופע
- שדות המחלקה מהווים מעין עצם עצמאי; השירותים שלו הם שירותי המחלקה; אבל לכל עצם מהמחלקה יש התייחסות אליו ומותר לשירותי המופע לגשת אליו
- לכן, לשדות המחלקה יהיה משתמר משלהם
- גם שירותי המחלקה וגם שירותי המופע חייבים לכבד אותו, כי לכולם יש גישה לשדות המחלקה

אתחול שדות המחלקה

- לפעמים המשתמר של שדות המחלקה פשוט ואתחול האוטומטי או הידני שלהם מבטיח את קיומו,

```
class Version {  
    private static Version free_list; null is ok  
    ...
```

```
class Paragraph {  
    public final static int DIR_LTR    =0x01;  
    ...
```

- אבל לפעמים זה לא מספיק; לפעמים המשתמר מסובך מדי, ולפעמים האתחול עלול להודיע על חריגים שצריך לטפל בהם

אתחול סטאטי

```
class Sentence {
    private static Set prepositions;
    static {
        prepositions = new HashSet();
        try {
            FileReader r
                = new FileReader("preps.txt");
            ... read the file and fill the set
        } catch (IOException e) {...}
    }
    ...
}
```

חוקי האתחול

- האתחול הסטאטי יתבצע לפני הפעלת שירות מהמחלקה, כולל שירותי מחלקה וכולל בנאים
- אם יש כמה גושי אתחול סטאטיים (ואולי גם פסוקי אתחול פשוטים) הם יתבצעו לפי סדר הופעתם; סגנונית, עדיף לרכז את כל האתחולים הסטאטיים לגוש אחד
- במחלקה מרחיבה שגם למחלקת הבסיס שלה יש אתחול סטאטי, האתחול של הבסיס יתבצע לפני האתחול של המרחיבה
- אם האתחול הסטאטי מפעיל שירותים של מחלקות אחרות, הן יאותחלו קודם
- סביבת זמן הריצה בוחרת בעצמה את זמן האתחול הסטאטי בהתחשב באילוצים הללו

singleton במקום אתחול סטאטי מסובך

- עדיף אולי להשתמש בתבנית היחיד (singleton) במקום בשדות מחלקה מרובים; מצב המחלקה נהפך למצב של עצם רגיל, שמשותף לכל העצמים במחלקה

```
class MyClassStatic {  
    ... instance fields  
    public MyClassStatic () {...}  
    ... instance methods  
}
```

```
class MyClass {  
    private final static MyClassStatic mcs  
        = new MyClassStatic();  
}
```

השתקפות (reflection)

- בג'אווה, המבנה של הקוד (מחלקות, שירותים, ושדות) זמין בזמן ריצה ומאפשר לחקור את מבנה הקוד
- יש מחלקה שהעצמים שלה מייצגים מחלקות, מחלקה שמייצגת חבילות, מחלקה לשירותים, מחלקה לשדות, ומחלקה לבנאים
- ניתן להפעיל בנאים ושירותים בעזרת העצמים המייצגים

`java.lang.Class`

`java.lang.Package`

`java.lang.reflect.Constructor`

`java.lang.reflect.Method`

`java.lang.reflect.Field`

המחלקה

```
VersionedString vs = ...
```

```
Class x = vs.getClass();           an Object method
```

```
Class y = VersionedString.class;   literal
```

```
Class z = Class.forName("VersionedString");  
static lookup
```

- העצם שמייצג את המחלקה יכול להחזיר את כל הפרטים לגביה: את שמה, את מי היא מרחיבה ומממשת, את השדות שלה, את השירותים והבנאים שלה

- למשל, בניית עצם תוך שימוש בבנאי ברירת המחדל:

```
VersionedString a  
= (VersionedString) z.newInstance();
```

בניית מחלקות באופן דינאמי

- המחלקה `java.lang.reflect.Proxy` מאפשרת לבנות מחלקות באופן דינאמי; אבל בדרך כלל יש דרך יותר פשוטה להשיג את אותה מטרה

```
class VSHandler implements
    InvocationHandler {
    public Object invoke(Object proxy,
                        Method m,
                        Object[] args) {
        if (m.getName().equals("add")) {...}
        else if (m.getName().equals("length"))
        ...
    }
}
```

למה זה טוב?

- בנייה דינאמית של עצמים כאלה שימושית כאשר מתקיימים שני תנאים
- ראשית, כאשר העצם הדינאמי הוא נציג (proxy) של עצם רגיל שאנו מבקשים להוסיף לו יכולת מסויימת
- את העצם הרגיל נעביר לבנאי של ה-`InvocationHandler` והשירות `invoke` יפעיל את השירות המבוקש על העצם הרגיל
- שנית, כאשר קבוצת השירותים של העצם הרגיל לא ידועה מראש, אלא ה-`InvocationHandler` מגלה אותה בזמן ריצה תוך שימוש בהשתקפות
- זה מאפשר, למשל, להוסיף שכבה של בדיקת הרשאות מול הגדרות בקובץ עבור אוסף שרירותי של מחלקות

סיכום שדות מחלקה והשתקפות

- שדות מחלקה משותפים לכל העצמים במחלקה
- לשדות המחלקה יש שמות גלובליים ידועים והם יחידים
- לשדות המחלקה יש משתמר משלהם; על כל השירותים של המחלקה, שירותי מופע ושירותי מחלקה, לכבד את המשתמר
- המחלקה היא לא רק אוסף הגדרות של שדות ושירותים, ולא רק מבנה הנתונים של שדות המחלקה
- המחלקה היא גם עצם מוחשי בזמן ריצה, עצם שניתן לחקור אותו ולקבל ממנו עצמים שמייצגים את השדות והשירותים
- אפשר להפעיל בנאים ושירותים בצורה כזו
- אפשר ליצור עצמים באופן דינאמי בעזרת `java.lang.reflect.Proxy`

חלק 9

נְשִׂיּוֹם (Naming)

מרחב השמות בתוכנית ג'אווה

- מרחב השמות בתוכנית ג'אווה, כפי שהצגנו אותו עד כה, הוא מרחב דו־שכבתי, כמעט שלם, עם חוקי נראות (visibility) דו־מימדיים

- שלמות: לכל דבר יש שם; ראינו שניתן להעביר לשירות עצם או מערך אנונימי, אבל לכל טיפוס היה עד כה שם

```
vi.add( new Integer (3) );  
printPrimes( new int[] { 1, 2, 3, 5, 7 } );
```

- דו־שכבתיות: אוסף של חבילות, שבכל אחת יש מחלקות; שם טיפוס מורכב משם החבילה ושם המחלקה; אוסף החבילות שטוח (למרות שהוא נראה היררכי) ואוסף המחלקות בחבילה שטוח

- דו־מימדיות: נראות מוחלטת, בחבילה, או ליורשים

בעצם מרחב השמות יותר מורכב

- יש לו יותר משתי שכבות: אפשר להגדיר מחלקות בתוך מחלקות, ואפילו מחלקות בתוך שירותים
- ואפשר גם ליצור מחלקות אנונימיות
- בחלק הזה של הקורס נראה את המנגנונים הללו
- וגם נגדיר בצורה מדוייקת את חוקי הנראות

מחלקות פנימיות סטאטיות

- הסוג הפשוט ביותר של מחלקה פנימית

```
public class PersistentVersionedString
    implements VersionedString {
    public static class PVSFilter
        implements java.io.FileFilter {
        public boolean accept(java.io.File f) {
            return f.getName().endsWith(".pvs");
        }
    }
    ...
}
```

שימוש במחלקה פנימית סטאטית

- מחלקה כזו היא מחלקה רגילה לכל דבר, פרט לזה ששמה המלא כולל את שם המחלקה החיצונית

```
FileFilter filter = new  
    PersistentVersionedString.PVSFilter();  
File dir = new File("/Projects/oopj");  
File[] files = dir.listFiles( filter );
```

- עצמים מהמחלקה הפנימית הם עצמאיים לחלוטין; אין שום קשר בינם ובין עצמים מהמחלקה החיצונית
- אבל בגלל שהפנימית היא מעין שדה של החיצונית, שירותים של שתיהן יכולים לגשת לכל שדות המחלקה של שתיהן, גם לשדות מוגנים (private ו-protected)

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 PersVS ... {  
    private static class PVSFilter ... {...}  
    public static FileFilter getFilter()  
        { return new PVSFilter(); }  
    ...  
}
```

```
FileFilter f = new PersVS.PVSFilter(); error
```

```
FileFilter f = PersVS.getFilter(); ok
```

עוד שימוש למחלקה פנימית

- Version הוא מחלקת עזר במימוש של `LinkedVersionedString`, עדיף להסתיר אותה

```
public class LinkedVersionedString
    extends VersionedString {
    private class Version {
        ...
    }
    ...
}
```

מחלקות פנימיות לא סטאטיות

- מבנה מיותר בג'אווה; מעט מאוד שימושים אמיתיים
- ובכל זאת, מה זה?
- מחלקה של עצמים שכל אחד מהם "שייך" לעצם של המחלקה המכילה ומכיר את שדות המופע שלו

```
public class Outer {  
    private int o;  
    public class Inner {  
        private int i;  
        public void set() { i = o; }  
        public int get() { return i; }  
    }  
}
```

קשירה של מחלקה פנימית

```
public class Outer {  
    private int o;  
    public class Inner {...}  
    public Inner getInner() {  
        return new Inner(); }  
    public void increment() { o++; }  
}
```

```
Outer x          = new Outer();  
Outer.Inner y1  = x.getInner();  
Outer.Inner y2  = x.getInner();
```

קשירה של מחלקה פנימית (המשך)

```
Outer x          = new Outer();  
Outer.Inner y1  = x.getInner();  
Outer.Inner y2  = x.getInner();
```

```
x.increment();  now x.o == 1  
y1.set();      y1.i = x.o == 1  
x.increment();  now x.o == 2  
y2.set();      y2.i = x.o == 2
```

```
y1.get();      returns 1  
y2.get();      returns 2
```

אותה תוצאה, מחלקה פנימית סטאטית

```
public class Outer {  
    public static class SInner {  
        private int i;  
        private Outer outer;           an explicit reference  
        public SInner(Outer outer) {  
            this.outer = outer; }  
        public void set() { i = outer.o; }  
        ...  
        public SInner getInner() {  
            return new SInner(this); }  
        ...  
    }  
}
```

תחביר (מסובך) לשימוש במח' פנימיות

- בנייה ישירה של עצם פנימי

```
Outer x = new Outer();
```

```
Outer.Inner y3 = x.new Outer.Inner();
```

- הרחבה של מחלקה פנימית על ידי מחלקה רגילה

```
class SubInner extends Outer.Inner {  
    public SubInner (Outer outer) {  
        outer.super(); } invoke the super's constructor
```

- שימוש בשדה מוסתר של המחלקה החיצונית על ידי הפנימית,
ובשדה מוסתר של המחלקה שהחיצונית מרחיבה

```
Outer.this.field
```

```
Outer.super.field
```

מחלקה מקומית (לא שימושי)

```
public VersionedString someMethod() {  
    final int fi = 3;  
        int mi = 4;  
class LocalVS implements VersionedString {  
    public void add(String s) {  
        int x = fi; ok  
        int y = mi; compilation error; mi is not final  
        ... }  
    ... }  
    return new LocalVS();  
}
```

תכונות מחלקה מקומית

- זו מחלקה פנימית
- לא סטאטית; אסור להשתמש במילת המפתח `static`
- הטיפוס לא מוכר מחוץ לגוש (שירות) שבו היא מוגדרת, ולכן אין צורך בהגדרת נראות (`private` או `protected`)
- מותר לה להשתמש במשתנים של הגוש (שירות) שבו היא מוגדרת; זו הסיבה להגדירה מקומית ולא סתם פנימית
- העצמים עצמם יכולים לחמוק מהגוש שבו המחלקה מוגדרת, ולכן מותר לה לגשת רק למשתנים שערכם לא ישתנה אחרי סיום פעולת הגוש (משתנים שמוגדרים `final`), אחרת היא הייתה עלולה לגשת למשתנים שכבר לא קיימים
- אין למחלקות כאלה הרבה שימושים

מחלקות אנונימיות (מאוד שימושיות)

- מחלקות אנונימיות הן בעצם הסיבה להגדרה של מחלקות פנימיות ומקומיות

- משמשות בדרך כלל לאריזה של פרוצדורה שמיועדת להישמר במבנה נתונים להפעלה בעתיד

```
Button b = new Button(...);
```

```
b.addMouseListener( new MouseListener() {  
    public void mouseClicked(Event e) {...}  
});
```

- העברנו לכפתור פרוצדורה שהוא אמור להפעיל כאשר לוחצים על הכפתור; הפרוצדורה הזו תגרום לתוצא הלוואי הרצוי; כדי לגרום לתוצא הלוואי, היא צריכה לשמור התייחסות לעצמים שהיא תשנה את מצבם; פרוצדורה כזו נקראת closure

שימוש טיפוס במחלקה אנונימית

```
interface MouseListener
{ public void mouseClicked(); }
class Button {
    Set mouse_listeners = new TreeSet();
    public void addMouseListener(
        MouseListener ml)
        { mouse_listeners.add(ml); }
    ...
}
b.addListener( new MouseListener() {
    public void mouseClicked(Event e) {...});
```

סיכום ביניים: מחלקות פנימיות

- הסוג השימושי ביותר הוא מחלקות אנונימיות, משום שהוא מפצה על היעדר התייחסויות לפרוצדורות בג'אווה
- מחלקות אנונימיות מאפשרות להעביר לעצם פרוצדורה כזו, שהתנהגותה תלויה בהקשר שבו הוגדרה (כי היא יכולה להשתמש בשדות של העצם ובמשתנים מקובעים של השירות שבו היא מוגדרת)
- מחלקות אנונימיות שימושיות בתבניות התיכון `observer`, `command` ו-`strategy`
- מחלקות פנימיות סטאטיות מאפשרות לעצב את מרחב השמות באופן גמיש ולהסתיר מחלקות עזר
- השאר (פנימיות לא סטאטיות ומקומיות) פחות שימושיות
- עדיף להגביל את התלות בין המחלקה החיצונית והפנימית

הגנה על שמות

- ארבע רמות הגנה על שמות: `protected`, `public`, `private` ו-`package` (בלי מילת מפתח)
- רק השמות מוגנים; העצמים עצמם לא
- ההגנה היא ביחס למבנה הסטאטי של הקוד, לא ביחס למבנה הדינאמי של עצמים בזיכרון: האם שורת קוד נתונה מסוגלת להתייחס לשם מסוים
- מה מוגן? מחלקות (כולל פנימיות) שדות, שירותים
- מה לא מוגן? משתנים ומחלקות בתוך שירותים/גושי פסוקים, טיפוסים אנונימיים (אין שם שאפשר להגן עליו)

רמות הגנה

- public: אין הגבלות
- private: שימוש רק על ידי קוד באותה מחלקה, כולל כל המחלקות פנימיות שמוגדרות באותה מחלקה ראשית (כולל מחלקות פנימיות אחרות)
- protected: כמו private אבל מתיר גישה למחלקות מרחיבות
- הגנת חבילה: כמו private אבל מתיר גישה לכל שירות באותה חבילה
- אפשר להגביל את התחום של הגנת חבילה על ידי "סגירת" חבילה כך שאחרים לא יוכלו להוסיף לה מחלקות; זה מתבצע על ידי סימון החבילה כחתומה (sealed) בקובץ ה-jar

חבילות ושמות חבילות

- מחלקות שוכנות בחבילות

- כל קובץ מזהה את החבילה שלה הוא שייך, ומיקום הקובץ צריך להתאים להיררכיה של שם החבילה

```
package il.ac.tau.oopj;
```

- למרות ששמות החבילות נראים היררכיים, ולמרות שקבצי קוד המקור והקבצים הבינריים (.java ו-.class) מאורגנים במדריכים באופן היררכי שמשקף את שמות החבילות, מבחינת הגנה על שמות אין היררכיה בין חבילות

- כלומר למחלקה בחבילה `il.ac.tau.oopj.ex3` אין גישה לשמות עם הגנת חבילה ב-`il.ac.tau.oopj` ולא להיפך

שמות מלאים ויבוא שמות

- השם המלא של חבילה כולל את שם החבילה ושם המחלקה, למשל, `java.io.InputStream`
- שימוש בשמות כאלה מקנה לקוד חד משמעיות, אבל הוא מסורבל, בייחוד כאשר שמות החבילות ארוכים ולאחר שם המחלקה בא שם שירות, או שם שדה, כמו למשל `java.lang.System.println`
- אפשר לייבא שמות מחלקות ספיציפיות או את כל שמות המחלקות מחבילה לקובץ קוד:

```
import java.io.InputStream;  
import java.lang.*;
```

- בג'אווה 1.5 ומעלה אפשר גם לייבא את שמות הקבועים שמוגדרים במחלקה (`static import`)

סיכום מרחב השמות

- מחלקות פנימיות מאפשרות להגדיר מרחב שמות היררכי שמסקף את מבנה הקוד (מי משתמש במי, מה שייך למה)
- מחלקות אנונימיות שימושיות בעיקר על מנת לייצג פרוצדורות קשורות חלקית (closures) במבני נתונים
- כדאי להימנע מתלויות סבוכות בין מחלקות חיצוניות ופנימיות ומתחביר לא טריוויאלי
- בחירת רמת ההגנה לשם דורשת בחירה בין יכולת שימוש והרחבה ובין מודולריות
- `protected` היא רמת ההגנה הבעייתית ביותר, כי אין לנו מושג מי ירחיב ומתי
- מחלקות פנימיות מאפשרות תיחום מודולריות יותר מדוייק