

חלק 8

מחלקות ושירותים מוכללים

אפשר בקלות להכליל, אבל

```
class VersionedObject {
    public int    length()      {...}
    public void   add(Object s) {...}
    public Object getVersion(int i) {...}
    public Object getLastVersion() {...}
}
```

- כעת אפשר לשמור גרסאות של כל דבר
 - אבל יש לנו שתי בעיות; ראשית, יתכן שכל גרסה תהיה מטיפוס אחר, וזה לא מה שהתכוונו,
- ```
vo.add("The letter A");
vo.add(new Integer(3));
```

### תוספת מאוחרת לג'אווה

- לא כל קומפיילר ג'אווה תומך בהכללה (generics)
- מחלקות ושירותים מוכללים נוספו לג'אווה רק בגרסה 1.5 שיצאה לאור בסתיו 2004. גרסאות קודמות לא תומכות במנגנון הזה. לתשומת לבכם אם אתם משתמשים בסביבת פיתוח ישנה.

### אובדן מידע לגבי טיפוסים

- בעיה שנייה היא שגם אם שמרנו רק גרסאות של מחרוזות, הערך המוחזר מ-`getVersion()` הוא מטיפוס `Object`; העובדה שהעצם ששמרנו היה מטיפוס `String` אבדה
- ```
VersionedObject vo = new VersionedObject();
vo.add("The letter A");
vo.add("The letter B");
String v1 = vo.getVersion(1); compilation error
String v2 = (String) vo.getVersion(2); ok
```
- אנו נאלצים לבחור בין קומפילציה ללא בדיקת טיפוסים מלאה (הבדיקה מתבצעת רק בזמן ריצה, ביציקה), ובין שכפול קוד (מחלקה `VersionedString` ומחלקה נפרדת `VersionedInteger`)

מחלקה ושירותים מוכללים (generic)

```
class Versioned<T> {
    public void add(T s)      {...}
    public int  length()      {...}
    public T    getLastVersion() {...}
    public T    getVersion(int i) {...}
}
```

- השם `T` מייצג טיפוס (מחלקה או ממשק)

ננסה להכליל את `VersionedString`

- השירותים שהמחלקה הגדירה (או הממשק והמחלקות השונות) כלל לא היו ספציפיים למחרוזות
- נניח שדרושה לנו מחלקה דומה עבור שלמים,

```
class VersionedInteger {
    public int    length()      {...}
    public void   add(Integer s) {...}
    public Integer getVersion(int i) {...}
    public Integer getLastVersion() {...}
}
```

- מימושי השירותים מ-`VersionedString` תקפים גם כאן
- אבל עדיף שלא לשכפלם

הכללה ויחס is-a

```
Versioned<String> vs =
    new Versioned<String> ();
Versioned<Object> vo =
    new Versioned<Object> ();
vo = vs; should this work?
vs.add("The letter A"); clearly allowed
vs.add(new Integer(3)); clearly not allowed
vo.add(new Integer(3)); oops, the compiler will
incorrectly allow this
```

- מסקנה: Versioned<String> הוא לא סוג של Versioned<Object>, זה לא אינטואיטיבי אבל נכון.

שימוש במחלקה מוכללת

```
Versioned<String> vs =
    new Versioned<String> ();
Versioned<Integer> vi =
    new Versioned<Integer> ();
vs = vi; compilation error
vs.add("The letter A");
vs.add("The letter B");
vs.add(new Integer(3)); compilation error
String v1 = vs.getVersion(1); ok, no need to cast
```

- השירות vs.add מצפה לקבל התייחסות מטיפוס T כאשר T הוא String, ובאופן דומה לגבי vs.getVersion

ולכן שירות כמו זה לא יעבוד כמצופה

• ננסה להגדיר שירות מוכלל (במחלקה לא מוכללת) שיבדוק האם בעצם עם גרסאות יש גרסאות עוקבות כפולות,

```
class VersionedUtils {
    static boolean consecutiveDuplicates(
        Versioned<Object> v) {
        for (int i=2; i<=v.length(); v++)
            if (v.getVersion(i).equals(
                v.getVersion(i-1))) return true;
        return false;
    } }

```

- כי אי אפשר לקרוא לו עם Versioned<String>

לפני הפרטים, העיקר

- מנגנון ההכללה מיועד לאפשר שימוש חוזר בקוד בלי לאבד מידע לגבי הטיפוס הסטטי של עצם
- בלי הכללה, שימוש חוזר בקוד מתבצע על ידי השמת התייחסות מטיפוס אחד לטיפוס אחר, יותר כללי; מאותו רגע אין דרך לשחזר את הטיפוס הסטטי המקורי בלי יציקה
- תפקיד ההכללה הוא למנוע צורך ביציקות, שנבדקות מאוחר
- בג'אווה, טיפוס מוכלל (כמו T) תמיד נקשר למחלקה/מנשק; לא למערך ולא לטיפוס פרימיטיבי
- הפרטים מסתבכים בגלל האינטראקציה בין מנגנון ההכללה ובין יחס הירושה (יחס ה-is-a)
- בשפות אחרות (C++) הכללה מיועדת גם לשיפור ביצועים

ג'וקרים

```
static boolean consecutiveDuplicates(
    Versioned<?> v) {
    for (int i=2; i<=v.length(); v++)
        if (v.getVersion(i).equals(
            v.getVersion(i-1))) return true;
    return false; }

```

- השימוש בג'וקר '?' מגדיר שירות (כאן פרוצדורה) שהארגומנט שלו הוא "סדרת גרסאות של משהו"
- פורמלית, Versioned<String> הוא לא סוג של Versioned<Object> אבל שניהם סוג של Versioned<?>

איך זה עובד

- הקומפיילר ממפה את כל המחלקות המוכללות Versioned<Something> למחלקה אחת רגילה (לא מוכללת) שהיא בעצם Versioned<Object>
- בקוד שמתמש במחלקה מוכללת, הקומפיילר מוסיף לקוד יציקות על מנת לבצע השמות מ-Object לטיפוס הספציפי, למשל String
- הקומפיילר מוודא שהיציקה תמיד תצליח ולעולם לא תודיע על ClassCastException

```
String v1 = (String) vs.getVersion(1);
```

- כלומר, הטיפוס המוכלל (T) נמחק מהקוד שהקומפיילר מייצר; הוא שימושי רק לבדיקות תקינות טיפוסים בזמן קומפילציה; התהליך נקרא מחיקה (erasure)

הגבלת הכלליות

- ננסה להגדיר פרוצדורה שבודקת האם הגרסאות בעצם הן בסדר מונוטוני עולה, (מבחינת התוכן, לא האינדקס)
 - למשל, הסדרות (1,3,4,898) ו-"X" ("A", "BCDE", "X") הן בסדר עולה, אבל (1,2,1,2,1) היא לא
- ```
static boolean monotonic(Versioned<?> v) {
 for (int i=1; i < v.length(); i++)
 if (v.getVersion(i).compareTo(
 v.getVersion(i+1)) < 1) error!
 return false;
 return true; }
```
- שגיאת קומפילציה - לא בטוח ש v הוא Comparable

### ננסה להשלים סדרת גרסאות

- ```
static void complete(Versioned<?> a,
                    Versioned<?> b) {
    for (int i = a.length()+1;
         i <= b.length(); i++)
        a.add( b.getVersion(i) );
}
```
- משלימים את סדרת הגרסאות הקצרה a כך שהסיפא של b תהיה הסיפא של a
 - **זה לא עובד!** שתי הסדרות הן גרסאות של משהו, אבל אולי לא של אותו משהו (הג'וקר יכול להיקשר בצורה שונה בכל אחד מהארגומנטים); את מה שמחזיר getVersion של b אולי אי אפשר להוסיף ל-a

איך מגבילים כלליות

- ```
static boolean monotonic
(Versioned<? extends Comparable> v) {
 for (int i=1; i < v.length(); i++)
 if (v.getVersion(i).compareTo(
 v.getVersion(i+1)) < 1) now it's ok
 return false;
 return true; }
```
- הגבלנו את הג'וקר: הוא לא יכול להחליף כל טיפוס, אלא רק טיפוס שהוא סוג של Comparable (כולל Comparable עצמו)

### הפתרון: שירות מוכלל

- ```
static <T> void complete(Versioned<T> a,
                       Versioned<T> b) {
    for (int i = a.length()+1;
         i <= b.length(); i++)
        a.add( b.getVersion(i) );
}
```
- השירות המוכלל מאלץ את שני המשהו-אים להיות אותו משהו, ולכן הערך המוחזר מ-b.getVersion הוא מטיפוס a.add לארגומנט של a.add
 - שירות מוכלל, כמו שירות של מחלקה מוכללת, יכול גם להחזיר ערך מטיפוס T ולהגדיר משתנים מקומיים מטיפוס T

לא הסתבכנו מדי?

- ברור שכן
- הרי גם בעזרת הכלים שהיו לנו לפני שהצגנו את מנגנון ההכללה יכולנו להגדיר את monotonic
- היינו מגדירים ממשק Versionable על מנת לייצג את האיברים של סדרת גרסאות, והיינו מגדירים אותו כממשק שמרחיב את Comparable, ואולי עוד ממשקים
- זה היה עובד, אבל אז היינו צריכים להשתמש ביציקות על האיברים ש-getVersion מחזיר מסדרה שמכילה רק מחרוזות או רק עצמים מטיפוס מסוים אחר
- כלומר **מטרת השימוש בהכללה היא למנוע יציקות**, אבל ברגע שמתחילים להשתמש בה צריך להשתמש בה גם בשירותים ומחלקות שבהן במקור לא הייתה שום בעיה

קריאה לשירות מוכלל

- ```
Versioned <String> vs1, vs2;
...
<String> complete(vs1,vs2);
 אבל ניתן להשמיט את הטיפוס ולכתוב
```
- ```
complete(vs1,vs2);
```
- כי הקומפילר יכול להסיק את הטיפוס מההקשר.

טיפוסים נאים (raw types)

- מנגנון ההכללה נוסף לג'אווה מאוחר, ולכן היה צורך לאפשר שימוש במחלקות פרמטריות גם מקוד ישן שאין בו הכללות

```
class Versioned<T> {...}
Versioned<String> vs =
    new Versioned<String>();
Versioned raw =
    new Versioned();           same as Versioned<?>
raw = vs;                      ok
vs = raw;                      "unchecked" compiler warning
```

- בשימוש בטיפוס נאי, פרמטר הטיפוס מוחלף בגבול העליון (בדרך כלל Object)

עוד על מנשקים בלי הכללה

- בגישה שמבוססת על מנשקים ללא שימוש בהכללה יש עוד בעיה
- String הוא סוג של Comparable, אבל לא סוג של Versionable
- המנשק Versionable כלל לא היה קיים כאשר הגדירו את המחלקה String
- גם אם הגדרת המנשק Versionable לא דורשת מאומה מעבר להרחבה (ריקה) של Comparable, עדיין הקומפיילר חושב ש-String הוא לא סוג של Versionable:

```
interface Versionable extends Comparable {}
```

מוזרויות

- בגלל שבג'אווה הכללה ממומשת באמצעות מנגנון המחיקה, בזמן ריצה אין זכר לפרמטר הטיפוס
- כלומר, בזמן ריצה אי אפשר להבחין בין עצם מטיפוס Versioned<String> ובין עצם מטיפוס Versioned<Integer>, ובפרט, בזמן ריצה נראה ששניהם מאותה מחלקה
- זה משפיע על בדיקת שייכות למחלקה (instanceof), על יציקות של עצמים מוכללים, ועל שדות המסומנים static (את המושגים הללו נבהיר כאשר נדון במחלקה כישות עצמאית)
- וזה מונע אפשרות לקרוא לבנאי על פי פרמטר טיפוס, כלומר

```
<T> void m(T x) { T y = new T(); ... } illegal
```

עוד דוגמה מעניינת

```
interface Comparator<T>() {
    static public int compare(T x, T y);
}

class Sorter<E> {
    static public
    int sort(E[] a, Comparator<...> c) {...}
}
```

- איך צריך להגדיר את פרמטר הטיפוס של Comparator (ההגדרה החסרה באדום)?

סיכום generics

- מנגנון ההכללה מאפשר להימנע מיציקות בלי לשכפל קוד
- קוד שאין בו יציקות מפורשות ושאינו בו טיפוסים נאים (ליתר דיוק, אם הקומפיילר לא הזהיר לגבי השימוש בטיפוסים נאים) הוא בטוח מבחינת טיפוסים (type safe)
- קוד כזה לא יכשל בביצוע יציקה בזמן ריצה: הבדיקות מועברות לזמן הקומפילציה
- השימוש בהכללה מסבך הצהרות על טיפוסים בגלל האינטראקציה הלא אינטואיטיבית בין טיפוסים מוכללים ובין יחס ה-is-a
- המימוש של הכללות בג'אווה כולל מספר מוזרויות

הגבלת כלליות עם חסם תחתון

- נניח, למשל, שפרמטר הטיפוס E קשור לטיפוס Double
- ברור שאם המשווה c יכול להשוות עצמים מטיפוס Double, שגרת המיון תוכל לפעול
- אבל יש עוד מקרים תקינים
- אם המשווה יכול להשוות עצמים מטיפוס Number, אז ברור שהוא יכול להשוות עצמים מטיפוס Double, כי Double הוא סוג של Number
- לפיכך, ברור שהמשוואה צריך להיות מסוגל להשוות עצמים מאיזשהו טיפוס T ש-Double הוא סוג שלו,

```
int sort(E[] a, Comparator<<? super E>) {...}
```

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; }
```

חלק 9

המחלקה פְּיִשׁוּת

בפני עצמה

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)

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() {...}
```

## 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 free\_list אחד אחד בתוכנית)
- זה בדיוק מה שרצינו: להיות מסוגלים למצוא את מאגר העצמים הפנויים, גם איננו מכירים אף עצם מהמחלקה, ושיהיה רק מאגר פנויים אחד

## 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

<p style="text-align: center;">קבועים</p> <ul style="list-style-type: none"> • שדות מחלקה משמשים גם לציון קבועים בתוכנית • מוסכמה סגנונית: בדרך כלל שמות שכוללים רק אותיות גדולות <pre>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) {...} ... }</pre>	<p style="text-align: center;">הערות לגבי הקצאת זיכרון</p> <ul style="list-style-type: none"> • 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 <pre>Version v1 = Version.alloc(); ... <i>we use the object that v1 refers to</i> v1.free(); Version v2 = Version.alloc(); v1.value = "something"; <i>an error!</i></pre> <ul style="list-style-type: none"> • בדרך כלל הפגם יותר קשה לגילוי מאשר בדוגמה
<p style="text-align: center;">שימוש בקבועים</p> <pre>class Paragraph { public final static int DIR_LTR =0x01; ... public Paragraph (int style) {...} ... } ... Paragraph p = new Paragraph(Paragraph.DIR RTL Paragraph.ALIGN_RIGHT);</pre>	<p style="text-align: center;">הקצאת זיכרון (ומשאבים אחרים)</p> <ul style="list-style-type: none"> • דליפת זיכרון (memory leak) הוא פגם אחר בניהול זיכרון מפורש: עצם מוקצה אבל לא משוחרר מפורשות (האם דליפת זיכרון היא פגם בדוגמה שלנו?) <pre>java.io.FileInputStream is = new FileInputStream("C:\grades.dat"); ... <i>we the input stream</i> is.close(); <i>we release the object; shouldn't use it later</i></pre> <ul style="list-style-type: none"> • 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
<p style="text-align: center;">המשתמר של שדות המחלקה</p> <ul style="list-style-type: none"> • (נניח את ההנחה הסבירה ששדות המחלקה מוגנים מגישה ישירה על ידי לקוחות של המחלקה ולכן אינם מופיעים בחזרה) • 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 	<p style="text-align: center;">עוד שימושים לשדות מחלקה</p> <ul style="list-style-type: none"> • יש עוד תבניות תיכון (design patterns) שמשתמשות 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 קונוי של מחרוזת

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();
}
```

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

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
```

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) {...}
    }
    ...
}
```

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();
```

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

חלק 10

נְיוּם (Naming)

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 שירותי של מחלקות

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

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`

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

עוד שימוש למחלקה פנימית

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

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

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");
        }
    }
    ...
}
```

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; }
    }
}
```

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/ooopj");
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 שתייהן יכולים לגשת לכל שדות המחלקה של שתייהן, גם לשדות מוגנים (protected ו-private)

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();
```

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
```

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();
}
```

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
```

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 להגדרה 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

אותה תוצאה עם 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); }
        ...
    }
}
```

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

### 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
```

<p style="text-align: center;">רמות הגנה</p> <ul style="list-style-type: none"> • 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 על ידי 	<p style="text-align: center;">שימוש טיפוזי במחלקה אנונימית</p> <pre>interface MouseListener { public void mouseClicked(); } class Button { Set mouse_listners = new TreeSet(); public void addMouseListener(MouseListener ml) { mouse_listeners.add(ml); } ... } b.addListener(new MouseListener() { public void mouseClicked(Event e) {...}});</pre>
<p style="text-align: center;">חבילות ושמות חבילות</p> <ul style="list-style-type: none"> • מחלקות שוכנות בחבילות • כל קובץ מזהה את החבילה שלה הוא שייך, ומיקום הקובץ צריך להתאים להיררכיה של שם החבילה <pre>package il.ac.tau.oopj;</pre> <ul style="list-style-type: none"> • למרות ששמות החבילות נראים היררכיים, ולמרות שקבצי קוד המקור והקבצים הבינריים (.class-ו .java) 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 	<p style="text-align: center;">סיכום ביניים: מחלקות פנימיות</p> <ul style="list-style-type: none"> • 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
<p style="text-align: center;">שמות מלאים ויבוא שמות</p> <ul style="list-style-type: none"> • 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 ספציפיות או את כל שמות המחלקות מחבילה לקובץ קוד: <pre>import java.io.InputStream; import java.lang.*;</pre> <ul style="list-style-type: none"> • בג'אווה 1.5 ומעלה אפשר גם לייבא את שמות הקבועים שמוגדרים במחלקה (static import) 	<p style="text-align: center;">הגנה על שמות</p> <ul style="list-style-type: none"> • 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, טיפוזים אנונימיים (אין שם שאפשר להגן עליו)

סיכום מרחב השמות

- מחלקות פנימיות מאפשרות להגדיר מרחב שמות היררכי שמשקף 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 מדויק