

חלק 6

הרחבת מחלקות

תוכנה 1 - חלק 6

הפתרון: מחלקה מופשטת

- צריך להשתמש במבנה תחבירי אחר, מחלקה מופשטת (abstract class)

```
abstract class VersionedString
    implements Comparable {
    public int compareTo(Comparable other)
        {...}
    abstract public void add(String s);
    abstract public int length();
    abstract public String getLastVersion();
    abstract public String getVersion(int i);};
```

4

תוכנה 1 - חלק 6

תזכורת: השיירות compareTo

```
int compareTo(Comparable other) {
    VersiondString other_vs
        = (VersiondString) other;
    if (this.length() > other_vs.length())
        return 1;
    if (this.length() < other_vs.length())
        return -1;
    return 0;
}
```

תוכנה 1 - חלק 6

2

מחלקה מופשטת

- כמו מנשק, מגדירה טיפוס לא שלם: אפשר להגדיר התייחסויות אליה אבל אי אפשר לייצר עצמים כאלה
- בניגוד למנשק, דרוש בנאי (או בנאי ברירת המחדל).
- מחלקה מופשטת אם לפחות שרות אחד מופיע כהצהרה (לא מימוש (חייבת להופיע מילת המפתח abstract מופשטת
- חייבים לציין גם בהגדרת המחלקה שהיא מופשטת
- מחלקה מופשטת מיועדת להיות מורחבת על ידי מחלקות מוחשיות שמגדירות את השיירות
- מנשק הוא למעשה מחלקה שכל שיירותיה מופשטים
- כמו בדיון על מנשקים, החחה מוגדר בדרך כלל על המחלקה המופשטת, לא על המחלקות שמממשות את כל השיירות

5

תוכנה 1 - חלק 6

שיכפול

- כזכור, פיתחנו את השיירות עבור המחלקה LinkedVersionedString, שמממש את המנשק VersionedString
- אותו שיירות בדיוק תקף לכל מחלקה שמממשת את המנשק
- צריך, אם כן, לשכפל את השיירות בכל מחלקה כזו
- השכפול לא רצוי: אם נגלה, למשל, פגם במימוש השיירות, נצטרך לתקן אותו בכל המחלקות שכוללות אותו
- רצוי היה להוסיף את הגדרת השיירות המשותף למנשק
- אבל מנשק לא יכול לכלול מימוש של שרות.

תוכנה 1 - חלק 6

3

הרחבת מחלקה מופשטת

```
class LinkedVersionedString
    extends VersionedString {
    protected int n;
    protected Version last;

    public void add(String s) {...}
    public int length() {...}
    public String getLastVersion() {...}
    public String getVersion(int i) {...}
}
```

- אין לכתוב כאן הגדרה לשיירות המשותף compareTo

6

תוכנה 1 - חלק 6

הרחבת מחלקה מוחשית

- ניתן להרחיב גם מחלקה מוחשית
- לדוגמא: רוצים לאפשר סימון גרסה בשם (מחרוזת) ולא רק במספר סודר
- לצורך המימוש צריך להוסיף שדה מופע, בנאי, ושירותים
- (אנחנו משתמשים במנשק `java.util.Map` ובמחלקה `java.util.TreeMap` שממשת אותו - פרטים בהמשך).
- (כזכור, המחלקה `Integer` היא הטיפוס העוטף של `int` שאנחנו משתמשים בו במקום `int` כאשר יש צורך בעצמים, למשל בהגדרת משתנים מטיפוס גנרי).

10

תוכנה 1 - חלק 6

ניראות visibility (סיכום)

- ארבע אפשרויות של ניראות של שדות ושירותים (וגם מחלקות מוכלות, שעדיין לא ראינו), בסדר עולה מהמצומצם לנרחב
- `private` (שדה או שרות) - נגיש רק לקוד באותה המחלקה (כולל האפשרות לגבי עצם אחר מאותה מחלקה).
- ניראות בחבילה (package visibility) [לא מופיע אף אחת מהמילים `public`, `private`, או `protected`] כנ"ל ובנוסף גם ממחלקות אחרות באותה חבילה.
- `protected` - כנ"ל ובנוסף גם מחלקות שירותות ממנה ושייכות לחבילה אחרת
- `public` - נגיש מכל מחלקה בלי הגבלה.
- ניראות בחבילה ו `public` ישימים גם לגבי מחלקה או מנשק

7

תוכנה 1 - חלק 6

```
import java.util.*;
class TaggedLinkedVersionedString
    extends LinkedVersionedString {
    protected Map <String,Integer> tags;

    public TaggedLinkedVersionedString() {...}
    public void tag(int i, String t) {...}
    public String getVersion(String t) {...}
}
```

11

תוכנה 1 - חלק 6

הרחבת מחלקה מופשטת

- המחלקה המרחיבה מספקת מימוש לשירותים המופשטים
- אם לא ניתן מימוש לכל השירותים, המחלקה המרחיבה היא בעצמה מופשטת, וחייבת לציין זאת.

```
abstract class Abst1 {
    abstract public void do1();
    abstract public void do2();
}
abstract class Abst2
    extends Abst1 {
    abstract public void do1() { .. }
}
```

8

תוכנה 1 - חלק 6

מבנה עצם מורחב

```
TaggedLinkedVersionedString
tags: ...
(LinkedVersionedString)
n: ...
last: ...
```

12

תוכנה 1 - חלק 6

הפעלת שירותים

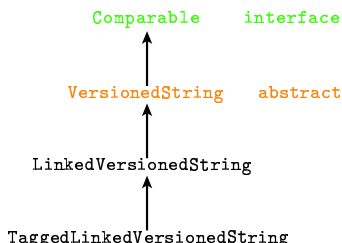
- כאשר מפעילים שירות על עצם ממחלקה שמרחיבה מחלקה מופשטת (יורשת ממנה), אם השירות מוגדר במחלקה המוחשית, הוא מופעל, אחרת מופעל השירות שנתקבל בירושה מהמחלקה המופשטת

```
VersionedString vs1 =
    new LinkedVersionedString();
VersionedString vs2 =
    new CyclicArrayVersString();
vs1.add("Mr. X");      LinkedVersionedString.add
int c =
    vs1.compareTo(vs2); VersionedString.compareTo
```

9

תוכנה 1 - חלק 6

היררכיית הטיפוסים



בנאי למחלקה המורחבת

```

public TaggedLinkedVersionedString() {
    super();
    tags = new TreeMap <String,Integer> ();
}
  
```

- קודם כל יש להפעיל את הבנאי של מחלקת הבסיס (המחלקה שאותה מרחיבים) בעזרת מילת המפתח super()
- אם למחלקת הבסיס יש מספר בנאים, אפשר להפעיל כל אחד מהם על ידי העברת ארגומנטים מתאימים, למשל, super("a demo");
- אם לא קוראים באופן מפורש לבנאי של מחלקת הבסיס, ג'אווה מכניסה אוטומטית את הקריאה super() לתחילת הבנאי

הסדר לא נכון

- מימוש השירותים ב- TaggedLinkedVersionedString לא תלוי בעצם ב- LinkedVersionedString
- אותו מימוש בדיוק יתאים גם ל- CyclicArrayVersionedString
- אפשר, למשל, להוסיף את השירותים הללו (ושדה המופע tags) למחלקה המופשטת VersionedString
- אבל אולי לא תמיד צריך את היכולות הללו, וכאשר לא צריך אותן, גם לא צריך את השדה tags
- כדאי אולי להפוך את הסדר

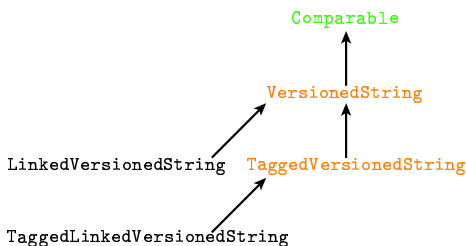
שירותים חדשים

```

/** attach the tag t to version no. i
 */
public void tag (int i, String t) {
    tags.put(t,i);
}

/** return the version tagged with t
 */
public String getVersion(String t) {
    return getVersion(tags.get(t));
}
  
```

נהפוך את הסדר



- כעת, אם המחלקה המוחשית מרחיבה את VersionedString היא לא מקבלת את השירותים הקשורים ב-tags, אבל אם היא מרחיבה את TaggedVersionedString, היא כן

המנשק MAP

- המנשק Map הוא חלק מ ה Java Collection Framework
- מייצג אוסף של מיפויים בין קבוצת מפתחות, לקבוצת עצמים.
- המנשק הוא גנרי. משתנה הטיפוס הראשון מייצג את המפתחות, והשני את הערכים הקשורים למפתחות.
- המפתחות שונים זה מזה. אם אינם מקובעים הלקוח צריך להבטיח שלא ישתנו כאשר הם בשימוש במיפוי.
- השירותים העיקריים הם
 - put (t, i) מוסיף מיפוי (או מחליף מיפוי קיים) מ t ל i
 - get (t) מחזיר את הערך שמקושר מהמפתח t
- TreeMap היא אחת המחלקות שמממשות את המנשק

הימנעות משכפול בלי ירושה מרובה

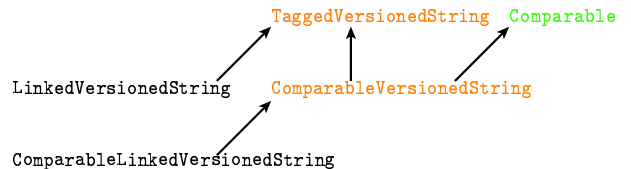
```
class VSComparator() {
    static a procedure, does not refer to a "this" object
    public int compare(VersionedString x,
        VersionedString y)
        {...}
}
class CLinkedVS extends LinkedVS
    implements Comparable {
    public int compareTo(VersionedString y) {
        return VSComparator.compare(this, y);
    }
}
```

22

תוכנה 1 - חלק 6

זה לא פתר את הבעיות

- המימוש של שתי המחלקות המוחשיות יכול להיות זהה לחלוטין; באחת יש tags ובשניה לא, תלוי את מי כל אחת מרחיבה; אם רוצים את שתיהן, מימוש השירותים משוכפל
- שנית, מה אם רוצים את השירותים הקשורים ב-tags, אבל לא את השירותים compareTo? אם זה המצב, צריך להפוך את הסדר של שתי המחלקות המופשטות



19

תוכנה 1 - חלק 6

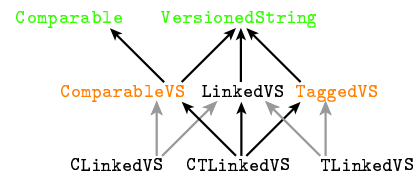
דוגמה יותר כללית

```
class Tagger() {
    protected java.util.Map tags;
    public Tagger() {...}
    public void tag(int i, String t) {...}
    public int get(String tag) {...}
}
class TLinkedVS extends LinkedVS
    implements Taggable {
    protected Tagger tagger = new Tagger();
    public void tag(int i, String t)
        { tagger.tag(i,t); }
}
```

23

תוכנה 1 - חלק 6

ירושה מרובה



- במבנה הזה כל מחלקה מוחשית יכולה לבחור את השירותים שהיא תספק על ידי ירושה ממחלקות מופשטות ומוחשיות מתאימות, ללא שכפול קוד
- למרבה הצער (?), ג'אווה לא מתירה ירושה מרובה (multiple inheritance, ירושה של שירותים מיותר מהורה יחיד בעץ הירושה)

20

תוכנה 1 - חלק 6

חס has-a במקום יחס is-a

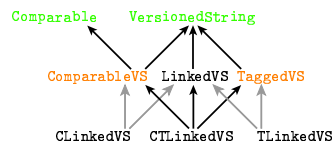
- ירושה מכלילה את שדות המופע של מחלקת הבסיס בעצם של המחלקה המרחיבה, מכלילה את השירותים, ומאפשרת להשתמש בעצם מרחיב כאילו היה עצם בסיס
- האפשרות לשימוש חלופי נקראת יחס is-a, למשל TLinkedVS is a LinkedVS
- הדוגמה הקודמת הראתה שניתן להשיג את אותן יכולות בדיוק על ידי הכלת עצם שלם ממחלקת הבסיס בעצם מהמחלקה המרחיבה (על מנת לקבל את שדות המופע של הבסיס), על ידי קריאה מפורשת לשירותי הבסיס (forwarding), ועל ידי הצהרה שהמחלקה מספקת את שירותי הבסיס (implements Taggable)
- יותר פשוט, יותר גמיש, אבל מעט יותר מסורבל

24

תוכנה 1 - חלק 6

למה אין ירושה מרובה בג'אווה

- למרות שירושה מרובה מאפשרת גמישות רבה בהגדרת מחלקות ללא שכפול קוד, המנגנון הזה יוצר בעיות
- נניח שגם ב-TaggedVS וגם ב-ComparableVS מוגדר שירות m
- כאשר מפעילים את CTLinkedVS, האם מופעל ComparableVS.m או TaggedVS.m? צריך מנגנון בחירה
- מנגנון הבחירה עשוי להשפיע גם על שירותים שמוגדרים גבוה יותר, למשל ב-VersionedString, אם זו מחלקה מופשטת ששירות שלה קורא לשירות m



21

תוכנה 1 - חלק 6

שיגור דינמי dynamic dispatch

ST
↑
DT1
↑
DT

- ניתוח סטטי של קריאה `target.method(...)` קובע מהו הטיפוס ST של המשתנה `target` (בין אם הוא שדה, משתנה מקומי או פרמטר) - הטיפוס הסטטי. (מנשק או מחלקה).
- בזמן ריצה, העצם ש `target` מתייחס אליו הוא מטיפוס שיושם מ ST. זהו הטיפוס הדינמי של `target`. DT.
- לכן רק בזמן ריצה ניתן לקבוע לאיזה שרות יש לקרוא: אם הוגדר שרות `method(...)` למחלקה DT, נקרא לו, אחרת יש לבדוק במחלקה ממנה ירש DT1, וכן הלאה.
- מובטח שיימצא שרות (אחרת שגיאת קומפילציה)
- הקומפילר יוצר קוד לביצוע הבחירה בזמן קבוע

28

תוכנה 1 - חלק 6

דריסת שירות (method overriding)

- גם אם מחלקה יורשת שירות מאחד ההורים הקדמונים שלה, היא יכולה להגדיר אותו מחדש
- ```
class LinkedVersionedString
 extends VersionedString {
 ...
 int compareTo(Comparable other) ... {
 VersionedString other_vs;
 ...
 if (this.n > other_vs.length())
 return 1;
 }
}
```
- יותר יעיל, התייחסות לשדה מופע n במקום קריאה לשירות

25

תוכנה 1 - חלק 6

## חיזוק החזקה

- בדוגמה רואים שימוש לחיזוק החזקה: אם לקוח צריך עצם ממחלקה עם חזקה משופר (CS), הוא צריך להשתמש במשתנה מטיפוס סטטי CS שמתייחס לעצם עם טיפוס דינמי CS
- זה נותן ללקוח אפשרות בחירה: חזקה רגילה דרך C, חזקה משופרת דרך CS
- למה שהלקוח יבחר בחזקה חלשה יותר? אולי המימוש של החזקה החלשה יותר יעיל, לפחות מההיבטים שחשובים ללקוח; למשל, הוספת איברים למערך לא ממוין יותר מהירה, למרות שחיפוש יותר איטי; אולי חשוב להשתמש במעט מחלקות על מנת למזער את מחיר בדיקות האיכות או גודל הזיכרון

29

תוכנה 1 - חלק 6

## שתי סיבות לדריסה

- אפשר לפעמים לממש את השירות יותר ביעילות במחלקה שמכירה את פרטי המימוש של העצם מאשר במחלקה מופשטת
- למשל, נניח שמחלקה C שומרת שלמים במערך לא ממוין; מחלקה חדשה CS יכולה לרשת ממנה אבל לדאוג שהמערך יהיה ממוין על ידי דריסת השירות שמוסיף איברים, ולחפש יותר ביעילות על ידי דריסת השירות שמחפש איברים
- סיבה שנייה: לפעמים היורשת יכולה לחזק את החזקה
- למשל, החזקה של האיטרטור של CS מחזירה מבטיח שהאיברים יוחזרו בסדר עולה, מכיוון שהמערך שמכיל אותם ממוין

26

תוכנה 1 - חלק 6

## החלשה או חיזוק?

- המנשק `VersionedString` והמחלקה `LinkedVersionedString` לא מגבילים את מספר הגרסאות שניתן לייצג, ולכן תנאי הקדם של `getVersion` הוא שהארגומנט יהיה בין 1 ובין `length()`, מספר הגרסאות שהוספו עד כה
- נניח שאנו רוצים להרחיב את המחלקה למחלקה חדשה `BoundedVersionedString` עם שירות נוסף, `chop(int i)`, שמוחק את הגרסאות `i-1` ומטה; אחרי פקודה כזו, התחום המותר של `getVersion` הוא רק עד `i` עד `length()`
- מבחינה מסוימת, חיזוקנו את החזקה, כי הוספנו שירות שלא היה קודם, אבל מבחינה אחרת, החלשנו את החזקה (את הספק), כי לספק החדש יש שירות עם תנאי קדם יותר מגביל

30

תוכנה 1 - חלק 6

## טיפוס סטטי ועצם דינמי

```
C static_C_dynamic_C = new C();
C static_C_dynamic_CS = new CS();
CS static_CS_dynamic_CS = new CS();
...
CIter i1 = static_C_dynamic_C.iterator();
// ברור שאי אפשר להסתמך על סריקה מונוטונית
CIter i2 = static_CS_dynamic_CS.iterator();
// ברור שכן אפשר להסתמך על סריקה מונוטונית
CIter i3 = static_C_dynamic_CS.iterator();
// האם אפשר להסתמך על סריקה מונוטונית?
```

27

תוכנה 1 - חלק 6

## עקרון ההחלפה (substitution)

- לקוח שמשמש בעצם דרך ייחוס מטיפוס `LinkedVersionedString` מניח שהעצם מקיים את החוזה של `LinkedVersionedString`
- הלקוח לא תמיד יכול לדעת מאיזה מחלקה העצם שאת שירותיו הוא מפעיל; אולי הוא מהמחלקה `LinkedVersionedString` ואולי הוא ממחלקה שמרחיבה אותה
- לכן עצם ממחלקה שמרחיבה את `LinkedVersionedString` חייב להתנהג ממש כאילו הוא מהמחלקה `LinkedVersionedString`
- כלומר צריך שאפשר יהיה להחליף אותו בעצם מהמחלקה `LinkedVersionedString` בלי שזה ישפיע על הלקוח

תוכנה 1 - חלק 6

31

## את החוזה של בנאי לא יורשים

- כי לא יורשים את הבנאי, אלא קוראים לו מפורשות מתוך בנאי במחלקה המרחיבה
  - בנוסף, לקוח תמיד קורא לבנאי של מחלקה מסוימת תוך שימוש בשם המחלקה, למשל
- ```
VersionedString bvs =  
new BoundedVersionedString();
```
- הלקוח לא מסתמך על החוזה של בנאי של מחלקת הבסיס
 - ולמנסקים אין בנאים כלל (למחלקה מופשטת יש בנאי - לצורכי המחלקות היורשות).

34

תוכנה 1 - חלק 6

ולכן אסור להרחיב את החוזה

- אסור לדרוש תנאי קדם יותר חזק (יותר קשה לקיום)
 - אסור להבטיח תנאי אחר יותר חלש
 - אסור להודיע על חריג מטיפוס שלא הוצהר במקור (זה נמנע תחבירית בג'אווה)
- במילים אחרות, החוזה של שירות דורס (או של שירות שמממש מנשק אבל עם חוזה מחוזק) הוא תמיד מהצורה
- תנאי הקדם הוא "תנאי קדם נורש או תנאי קדם אחר"
- תנאי האחר הוא "תנאי אחר נורש וגם תנאי אחר נוסף"
- כמובן שמותר לחזק רק תנאי קדם (תנאי אחר נוסף `true`) או רק תנאי אחר (תנאי קדם נוסף `false`)

תוכנה 1 - חלק 6

32

תכנון המשפחה

- מחלקה `Sub` שמרחיבה מחלקה `Super` (או מממשת מנשק `Super`) יכולה להשתמש בחוזה של `Super` ללא שינוי,
- או שהיא יכולה לחזק אותו, כלומר לדרוש פחות מהלקוח ו/או להבטיח לו יותר
- אבל אסור ל-`Sub` להשתמש בחוזה שהוא חלש משל `Super`
- לכן, אם מתכננים היררכיה, המחלקות עם החוזים המגבילים יותר צריכות להיות למעלה, והמחלקות המתירניות למטה (להרחיב את המגבילות, ולא להיפך)
- בפרט, היררכיה של מחלקות צריך לתכנן, וגם
- אי אפשר להשתמש במחלקות קיימות כבסיס למחלקות חדשות שרירותיות אם יוצרים את החדשות על ידי ירושה

35

תוכנה 1 - חלק 6

אם עוד לא השתכנעתם ...

```
public void doIt(VersionedString vs) {  
    for (int i=1; i<=vs.length(); i++) {  
        String s = vs.getVersion(i);  
        do something with s  
    }  
}
```

```
VersionedString bvs =  
    new BoundedVersionedString();  
add versions to bvs  
doIt(bvs);
```

תקין תחבירית אבל מפר את עקרון ההחלפה

תוכנה 1 - חלק 6

33

העמסה וירושה

- זה נראה סביר (הפרוצדורות מתוך `java.lang.String`):
- ```
static String valueOf(double d) {...}
static String valueOf(boolean b) {...}
```
- אבל מה עם זה?
- ```
overloaded(VersionedString x) {...}  
overloaded(LinkedVersionedString x) {...}
```
- לא נורא, הקומפיילר יכול להחליט,
- ```
VersionedString vs = new LinkedVS();
LinkedVersionedString lvs = new LinkedVS();
overloaded(vs); We must use the more general method
overloaded(lvs); The more specific method applies
```

36

תוכנה 1 - חלק 6

## אבל זה כבר מוגזם

```
overTheTop(VS x, LinkedVS y) {...}
overTheTop(LinkedVS x, VS y) {...}
```

```
LinkedVS a = new LinkedVS();
LinkedVS b = new LinkedVS();
overTheTop(a, b);
```

- ברור שצריך יציקה אחת למעלה, אבל איזו מהן, על a או b?
- אין דרך להחליט; הפעלת השגרה לא חוקית בג'אווה

תוכנה 1 - חלק 6

37

## מה ייבחר

- מנגנון ההעמסה הוא סטטי: בוחר את החתימה של השרות (טיפוס העצם, שם השרות, מספר וסוג הפרמטרים), אבל יעדין לא קובע איזה שירות ייקרא. עבור הקריאה  
(B) o).overloaded( lvs )
- תיבחר החתימה B.overloaded(VS) בגלל שיעד הקריאה הוא מטיפוס B (השרות היחיד הרלבנטי הוא האדום!)
- בזמן ריצה מופעל מנגנון השיגור הדינמי, שבוחר בין השרותים בעלי חתימה זאת, את המתאים ביותר, לטיפוס הדינמי של יעד הקריאה. הטיפוס הדינמי הוא S, לכן נבחר השרות הירוק.
- כנ"ל אם הקריאה היא  
B b = new S(); b.overloaded( lvs )

תוכנה 1 - חלק 6

40

## שברירות

```
overTheTop(VS x, LinkedVS y) {...}
overTheTop(LinkedVS x, VS y) {...}
```

```
LinkedVS a = new LinkedVS();
LinkedVS b = new LinkedVS();
overTheTop(a, b);
```

- אם הייתה רק הגרסה הירוקה, הקריאה לשגרה הייתה חוקית
- כאשר מוסיפים את הגרסה הכתומה, הקריאה נהפכת ללא חוקית; אבל הקומפילר לא יגלה את זה אם זה בקובץ אחר, והתוכנית תמשיך לעבוד, ולקורא לגרסה הירוקה
- לא טוב שקומפילציה רק של קובץ שלא השתנה תשנה את התנהגות התוכנית; זה מצב שברירי

תוכנה 1 - חלק 6

38

## אם עוד לא השתכנעתם

- שהעמסה היא רעיון מסוכן,
- אז עכשיו זה הזמן
- בייחוד כאשר ההעמסה היא ביחס לטיפוסים שמרחיבים זה את זה, לא זרים לחלוטין
- יוצר שברירות, קוד שמתנהג בצורה לא אינטואיטיבית (השירות שעצם מפעיל תלוי בטיפוס ההתייחסות לעצם ולא רק במחלקה של העצם), וקושי לדעת איזה שירות בדיוק מופעל
- ומכיוון שהתמורה היחידה (אם בכלל) היא אסתטית, לא כדאי

תוכנה 1 - חלק 6

41

## אולי יותר גרוע

```
class B {
 overloaded(VS x) {...}
}
class S extends B {
 overloaded(VS x) {...} override
 overloaded(LinkedVS x) {...} override but
 no override!
}
S o = new S(); LinkedVS lvs = ...
o.overloaded(lvs); invoke the orange
(B) o).overloaded(lvs); What to invoke?
```

תוכנה 1 - חלק 6

39

## עקרון הפתוח-סגור

- המרכיבים של מערכת תוכנה צריכים להיות סגורים, כלומר מוכנים לשימוש, ובה בשעה להיות פתוחים, מוכנים להרחבה ללא שינוי הקיים
- הרחבה על ידי שינוי הקיים פחות רצויה (אם כי ניתן לבצע refactoring - שינוי מבני שאינו משנה התנהגות).
- מנגנון הירושה הוא מימוש אחד של העיקרון; זה לא המימוש היחיד (גישה אחרת למימוש - plugins)
- בג'אווה, מימוש העיקרון על ידי ירושה אינו שלם
- אי אפשר לרשת מיותר מהורה אחד
- אי אפשר להרחיב אם ההרחבה יותר מגבילה מהבסיס; דרוש מנגנון שמאפשר לרשת מימוש בלי שיוצר יחס בין הטיפוסים

תוכנה 1 - חלק 6

42

## החיים במשפחה מורחבת

- עד כה דנו ביחס שבין מחלקה מרחיבה ובין לקוחות; לקוחות שלה, לקוחות של מחלקת הבסיס או המנשק, ולקוחות שאינם יודעים בדיוק באיזו מחלקה הם משתמשים (הרוב)
- כעת נדון ביחס שבין מימוש מחלקה ובין המימוש של הרחבות שלה
- המחלקות לאורך מסלול בהיררכיית הטיפוסים הן מעין משפחה מורחבת שיש לה כללים שיש לשמור עליהם
- לעומת זאת, על המימושים של מחלקות שאינן על מסלול כזה אין שום הגבלות, גם אם כולן ממשות את אותו מנשק

תוכנה 1 - חלק 6

43

## אבל יש כאן חוסר סימטריה

- מהמחלקה המרחיבה אפשר לדרוש שתכבד את המשתמר של מחלקת הבסיס; היא נכתבת מאוחר יותר ומתייחסת מפורשות למחלקת הבסיס
- אבל מחלקת הבסיס לא מודעת להרחבות שלה, וייתכנו הרחבות רבות, ואי אפשר לדעת מתי תתווסף הרחבה חדשה
- לכן אי אפשר לדרוש ממחלקת בסיס לכבד את המשתמרים של המחלקות שמרחיבות אותה
- במקום זה, נדרוש שהמשתמר של המחלקה המרחיבה יוגדר כך ששירותים לא דרושים של מחלקת הבסיס לא יפרו אותו
- איך דואגים לכל זה?

תוכנה 1 - חלק 6

46

## שירותים מעורבים

```
class TaggedLinkedVersionedString
 extends LinkedVersionedString {...}

VersionedString vs =
 new TaggedLinkedVersionedString();
vs.add("Mr. X"); LinkedVersionedString.add
vs.add("Ms. Y"); LinkedVersionedString.add
vs.tag(2, "F"); TaggedLinkedVersionedString.add
```

- אלא אם דרסנו את כל השירותים, שירותים ממחלקת הבסיס ומהמחלקה המרחיבה מופעלים באופן מעורב

תוכנה 1 - חלק 6

44

## כיצד לכבד משתמרים

- המקרה הפשוט ביותר: שירותי המחלקה המרחיבה לא משנים את שדות המופע של מחלקת הבסיס, והמשתמר של המחלקה המרחיבה לא מתייחס כלל לשדות המופע של מחלקת הבסיס
- ההימנעות משינוי שדות המופע של הבסיס מבטיחה ששירות של המרחיבה לא יפר את המשתמר
- וההימנעות מהתייחסות לשדות המופע הללו במשתמר של המחלקה המרחיבה מבטיחה, בדרך כלל, מהפרה של המשתמר הזה על ידי שירות של מחלקת הבסיס
- מקרה פשוט אחר, פחות נפוץ: כל השירותים נדרסים, שירותים של שתי המחלקות לא מופעלים לסירוגין

תוכנה 1 - חלק 6

47

## ולכן,

- שירותי המחלקה המרחיבה חייבים לכבד את המשתמר של המחלקה המורחבת, מכיוון ששירות של המחלקה המורחבת עלול להיקרא אחרי שירות מהמחלקה המרחיבה
- ולהיפך, שירות של המחלקה המרחיבה עשוי להיות מופעל אחרי שירות של המחלקה המורחבת, ולכן גם המשתמש של המורחבת חייב להתקיים אחרי סיום פעולת שירות של מחלקת הבסיס
- כללית, המשתמר של המחלקה המרחיבה יכול להיות חזק יותר, אך לא חלש יותר.

תוכנה 1 - חלק 6

45

## חזרה לדוגמא Stack

- השאילתה `full()` שהוספנו למנשק המחסנית נועדה לאפשר לבטא כבר במנשק תנאי קדם שהיו לכאורה רלבנטיים רק לאחד המימושים `FixedCapacityStack`
- כתוצאה מזה גם `LinkedStack` נדרש לספק מימוש לשאילתה הזאת (מחזיר תמיד `false`)
- לא תמיד נחזה מקרה כזה מראש, בזמן תכנון המנשק, וייתכן שנאלץ להוסיף מאוחר יותר.

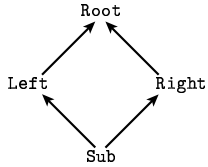
תוכנה 1 - חלק 6

48



## זה מעורר עוד סוגיה בירושה מרובה

- האם לשתף אב קדמון או לא?



52

תוכנה 1 - חלק 6

## שימוש בשירותים דרוסים

- לפעמים צריך או רצוי להשתמש בשירות דרוס מתוך השירות הדורס, למשל,

```
class AudibleButton extends Button {
 public void Clicked() {
 getDisplay().play(clickSound);
 super.Clicked();
 }
}
```

- השירות הדורס מחזק את החוזה של הנדרס על ידי הוספת תוצא לוווי לנדרס; דוגמאות אחרות מחזקות את תנאי האחר
- השירות הדורס אחראי לקיום תנאי הקדם של הנדרס ולקיום המשתמר של מחלקת הבסיס בנקודה שבה הנדרס נקרא

49

תוכנה 1 - חלק 6

## לשתף את האב הקדמון?

- אם Left ו-Right משתמשות כל אחת בשדה field של Root באופן שרירותי, ברור כל אחת מהן צריכה עותק פרטי שלו
- ומה אם שתיהן מכבדות את המשתמר של Root?
- זה לא תמיד מספיק; נניח שהמשתמר של Root הוא  $i > 0$ , ש-Left מוסיפה למשתמר את התנאי  $i > 5$ , אבל Right מוסיפה  $i < 4$ , וש-Root לא מפירה את המשתמרים הללו
- ברור ששיתוף יכשל
- אי אפשר לפתור את זה ב-Left ו-Right כי אין בניהן שום יחס
- הימיוש של Sub צריך לבחור האם לשתף; העסק מסתבך

53

תוכנה 1 - חלק 6

## שדות מופע לא נדרסים

- הם מוסתרים (shadowed); ככלל, זה מבלבל ולא רצוי
- אם למחלקת הבסיס B יש שדה מופע f, וגם המחלקה המרחיבה S מצהירה על שדה מופע f, בעצמים מהמחלקה S יהיו שני שדות מופע שונים בשם f, אחד של B אחד של S
- ההתייחסות לשדות מופע היא סטאטית: שירותים שמוגדרים ב-B מתייחסים לשדה המופע של B (למרות שהעצם הוא בפועל מהמחלקה S) ושירותים שמוגדרים ב-S לשדה המופע של S
- זה שונה לגמרי מההתייחסות לשירותים: תמיד לפי הטיפוס הדינמי
- ניתן לבחור את שדה המופע על ידי `super.f` בשירותים של S או על ידי יציקה `((B) this).f`

50

תוכנה 1 - חלק 6

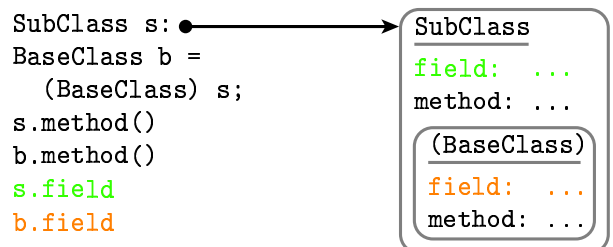
## מה עוד לא יורשים

- גם שרותי מחלקה מוסתרים ולא נדרסים (ואם נזכור לקרוא להם באמצעות שם מחלקה, ולא עצם, אזי השם המלא של השרות שונה, אין הסתרה).
- אסור לשרות מחלקה במחלקה יורשת להסתיר שרות מופע במחלקת הבסיס.
- אין יורשים שדה או שרות פרטי. אם מופיע שדה או שרות באותו שם במחלקה היורשת, אין לו כל קשר לזה שבמחלקת הבסיס.

54

תוכנה 1 - חלק 6

## ההתייחסות סטאטית לשדות מופע



51

תוכנה 1 - חלק 6

## דוגמה נוספת - call back

- שרות מוחשי במחלקה מופשטת יכול לקרוא לשירות מופשט.
  - מנגנון של call back (קוד "מערכת" מפעיל קוד משתמש)
- ```
abstract class Stack <T> {
    public void change_top(T t) {
        pop (); push(t);
    }
    abstract public void push(T t);
    abstract public void pop();
}
```

58

תוכנה 1 חלק 6

מנגנון ההרחבה פוגע במודולריות

- הרעיון המרכזי שעמד מאחורי עצמים ומאחורי חוזים היה מודולריות: היכולת להפריד את ההיבטים של מחלקה שרלוונטיים ללקוחות מההיבטים שהם עניינה הפרטי
- זה מאפשר לשנות את המימוש של מחלקה בלי להשפיע על לקוחות, ובפרט להפריד את פיתוחה ותחזוקתה
- אם מאפשרים הרחבה וחושפים את המימוש למחלקות מרחיבות (הגנה על שדות המופע ברמת `protected`), שינויים במימוש עשויים להשפיע על מחלקות מרחיבות
- לפעמים זה לא נורא, אם המחלקות המרחיבות ממומשות ומתוחזקות על ידי אותו צוות פיתוח, או אם יש ביטחון בכך שהמימוש של מחלקת הבסיס מוצלח ולא ישתנה בעתיד
- אבל הרחבות שוברות מודולריות זה עלול להיות נורא

55

תוכנה 1 חלק 6

שינוי הטיפוס בירושה

- בג'אווה 5 שרות דורס יכול לשנות את טיפוס הערך המוחזר באופן קו-וריאנטי. כלומר טיפוס הערך המוחזר של השרות במחלקה המרחיבה צריך להיות הרחבה של הטיפוס במחלקת הבסיס (או שווה לו). מדוע זה בטיחותי?
- ```
public class B {
 public B do1(...) { ... }
 public B1 do2(...) { ... }
}
public class S extends B {
 public S do1(...) { ... }
 public S1 do2(...) { ... }
 // S1 must inherit from B1
}
```

59

תוכנה 1 חלק 6

## גישות לשימור המודולריות

- שימוש ב-`has-a` במקום ב-`is-a`, כלומר שימוש בשדה מופע במקום בירושה
- שימוש ברמת הגנה `private` עבור שדות המופע של מחלקת הבסיס והשירותים הלא ציבוריים שלה; זה מספק רמת הגנה דומה לשימוש ב-`has-a`
- איסור מוחלט להרחיב את המחלקה על ידי שימוש במילת המפתח `final` בהגדרת המחלקה
- (מילת המפתח `final` בהגדרת שירות אוסרת לדרוס אותו, אבל שירותים אחרים של מחלקה מרחיבה עשויים להיות תלויים במימוש מחלקת הבסיס; כלומר זה לא משמר מודולריות)

56

תוכנה 1 חלק 6

## הערות על בדיקות של היררכיית טיפוסים

- לגבי מחלקות שמממשות מנשק: יש לבצע בדיקת קופסה שחורה מול החוזה של המנשק (אם לא חזקנו אותו) וכיסוי של המימוש
- לגבי מחלקות שמרחיבות מחלקות: דרושה בדיקה מלאה של מחלקת הבסיס, בדיקה של השירותים הנוספים/מחזוקים של המרחיבה, ובדיקת כיסוי של המרחיבה
- מחלקה מופשטת צריך לבדוק בעזרת מחלקה מוחשית מרחיבה

60

תוכנה 1 חלק 6

## סוף הדרך

- אם מטפסים ממחלקה כלשהי בגרף שיחס ה-`extends` מגדיר, מגיעים תמיד למחלקה `java.lang.Object`
- כלומר הגרף הוא עץ מכוון עם שורש
- אם הגדרה של מחלקה לא מציינת את מי היא מרחיבה, השפה מוסיפה אוטומטית את פסוק ההרחבה `extends java.lang.Object`
- זה מבטיח שכל עצם הוא סוג של `Object`
- זה מבטיח שכל עצם בג'אווה מספק שירותים בסיסיים מסוימים, אם על ידי ירושה מ-`Object` או על ידי דריסה
- השירותים הללו הם `toString`, `clone`, `equals`, ועוד כמה פחות חשובים

57

תוכנה 1 חלק 6

## **סיכום הרחבה וירושה**

- מאפשרת להוסיף למחלקה שדות מופע ושירותים או לדרוס שירותים ולהגדיר אותם מחדש
- מאפשרת שימוש נוסף בקוד ללא שכפול, ההרחבה ותיקון של מחלקות שאין את קוד המקור שלהן
- למחלקה מרחיבה או מחלקה שמממשת מנשק אבל משנה את החוזה שלו אסור להחליש את החוזה, אבל שפת התכנות לא כופה את האילוף הזה
- ירושה כפולה (אין בג'אווה) מאפשרת יותר גמישות ביצירת מחלקות חדשות, אבל קשה להשתמש בה
- שימוש עצם כשדה מופע במקום לרשת אותו מאפשר לנצל שירותים ממספר מחלקות (במקום ירושה כפולה) ולהפריד לחלוטין את גרף ה-is-a מגרף התלויות בין מימושים