# חלק 6

# הרחבת מחלקות

# תזכורת: השירות 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;
}
```

# שיכפול

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

# 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);
}
```

# 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 ללא מימוש (חייבת להופיע מילת המפתח `abstract`)
- חייבים לציין גם בהגדרת המחלקה שהיא מופשטת
- מחלקה מופשטת מיועדת להיות מורחבת על ידי מחלקות מוחשיות שמגדירות את השירותים המופשטים
- מנשק הוא למעשה מחלקה שכל שירותיה מופשטים
- כמו בדיון על מנשקים, החוזה מוגדר בדרך כלל על המחלקה המופשטת, לא על המחלקות שממשות את כל השירותים

# 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 לשירות המשותף compareTo

# נירות visibility

ארבע אפשרויות של נירות של שדות ושירותים (וגם מחלקות מוכלות, שעדיין לא ראינו), בסדר עולה מהמצומצם לנרחב

- `private` (שדה או שרות): נגיש רק לקוד באותה המחלקה (כולל דרך עצם אחר מאותה מחלקה)
- נירות בחבילה (package visibility): לא מופיע אף אחת מהמילים `public`, `private`, או `protect` , כנ"ל ובנוסף גם ממחלקות אחרות באותה חבילה
- `protected`: כנ"ל ובנוסף גם מחלקות שיורשות ממנה ושייכות לחבילה אחרת
- `public`: נגיש מכל מחלקה בלי הגבלה
- מחלקה או מנשק (לא מוכלים) צריך להגדיר עם נירות בחבילה או `public`

# 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 {
  public void do1() { ... }
}
```

# 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
```

# הרחבת מחלקה מוחשית

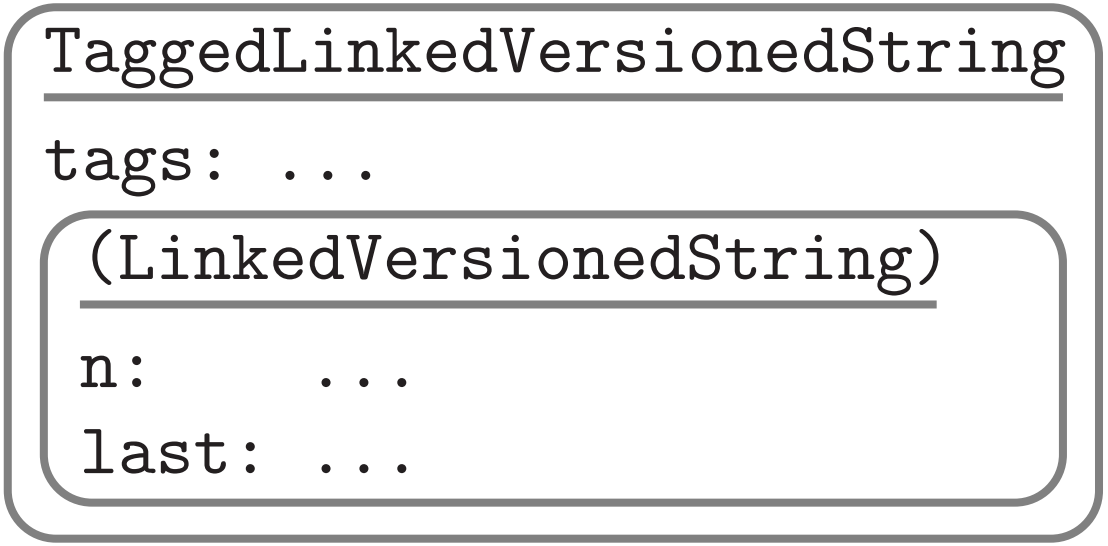
- ניתן להרחיב גם מחלקה מוחשית
- לדוגמה: רוצים לאפשר סימון גרסה בשם (מחרוזת) ולא רק במספר סודר
- לצורך המימוש צריך להוסיף שדה מופע, בנאי, ושירותים
- (אנחנו משתמשים במנשק `java.util.Map` ובמחלקה `java.util.TreeMap` שממשת אותו; פרטים בהמשך)
- (כזכור, המחלקה `Integer` היא הטיפוס העוטף של `int` שאנחנו משתמשים בו במקום `int` כאשר חייבים להשתמש בעצם ולא בטיפוס פרימיטיבי

```
import java.util.*;

class TaggedLinkedVersionedString
      extends LinkedVersionedString {
  protected Map tags;

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

# מבנה עצם מורחב



# בנאי למחלקה המורחבת

```
public TaggedLinkedVersionedString() {
  super();
  tags = new TreeMap();
}
```

- קודם כל יש להפעיל את הבנאי של מחלקת הבסיס (המחלקה שאותה מרחיבים) בעזרת מילת המפתח `super`
- אם למחלקת הבסיס יש מספר בנאים, אפשר להפעיל 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

# שירותים חדשים

*attach the tag t to version no. i*

```
public void tag (int i, String t) {
  tags.put(t,i);
}
```

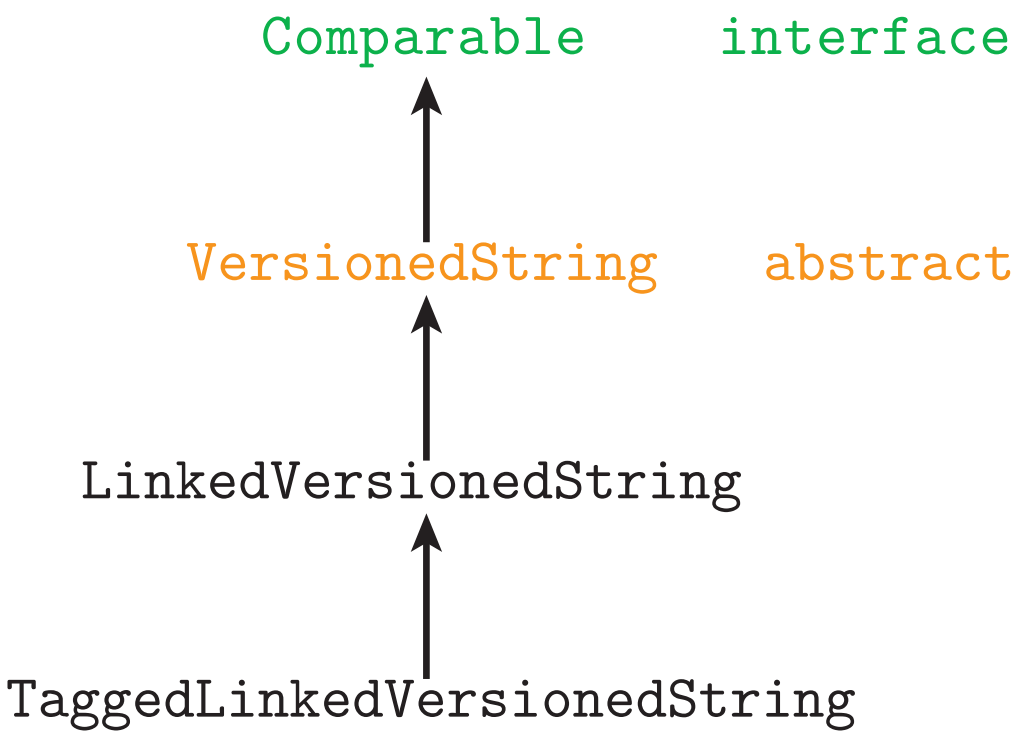
*return the version tagged with t*

```
public String getVersion(String t) {
  return (String) getVersion(tags.get(t));
}
```

# המנשק Map

- המנשק `Map` הוא חלק מה-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 שממשות את המנשק

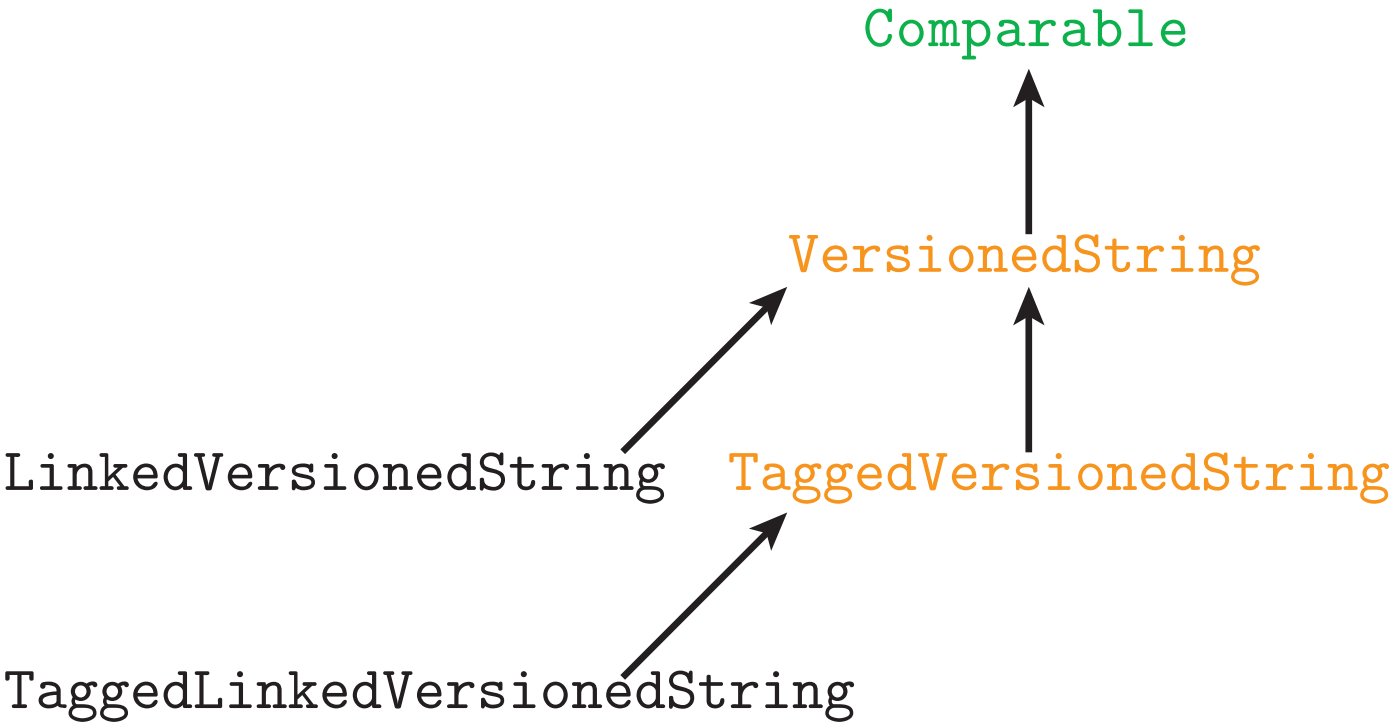
# היררכיית הטיפוסים



# הסדר לא נכון

- מימוש השירותים ב-
  `TaggedLinkedVersionedString` לא תלוי בכלל ב-
  `LinkedVereionedString`
- אותו מימוש בדיוק יתאים גם ל-
  `CyclicArrayVersString`
- 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

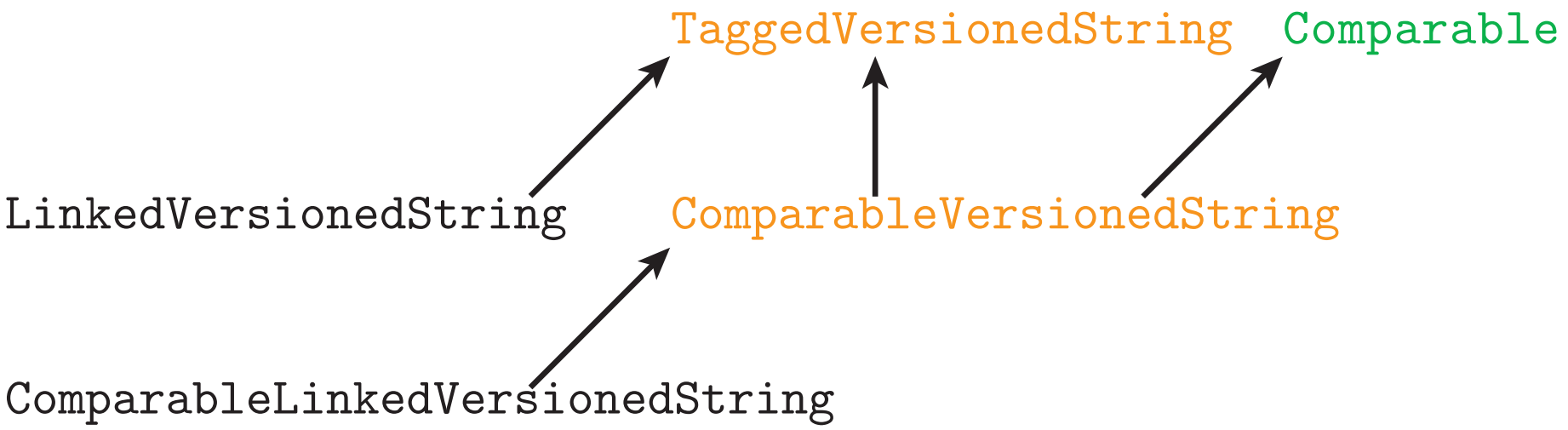
# נהפוך את הסדר



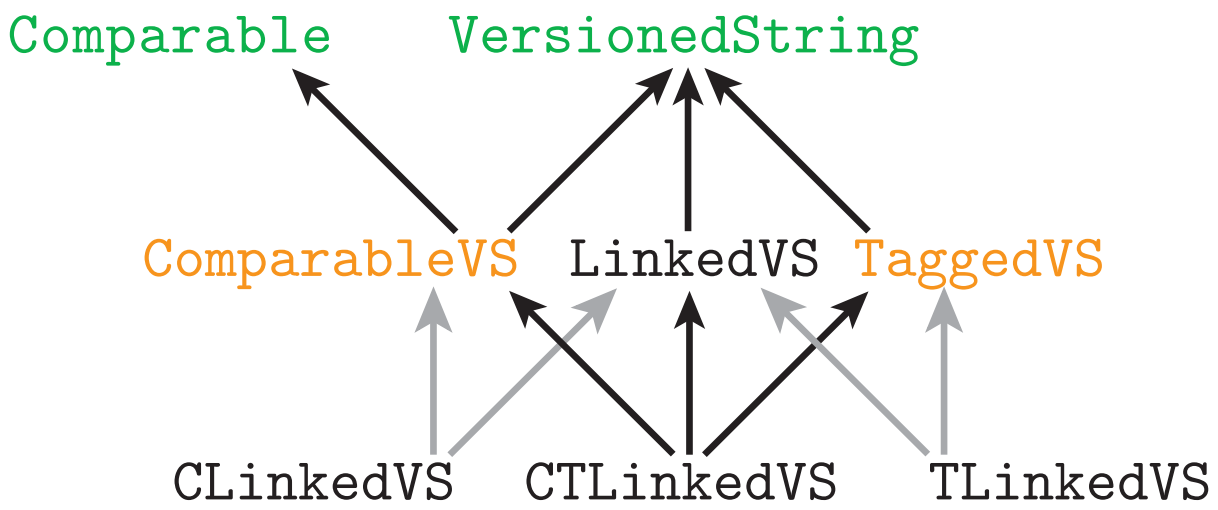

- כעת, אם המחלקה המוחשית מרחיבה את VersionedString היא לא מקבלת את השירותים הקשורים ב-tags, אבל אם היא מרחיבה את TaggedVersionedString, היא כן

# 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 השירות `compareTo`? אם זה המצב, צריך להפוך את הסדר של שתי המחלקות המופשטות



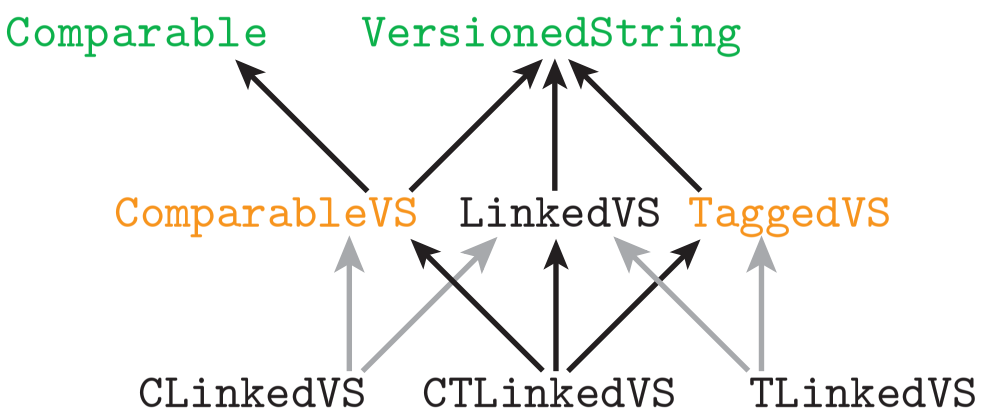
# ירושה מרובה




- 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)

# 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



# 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);
 }
```

# 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); }
```

# יחס 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`)
- מבנה יותר פשוט, יותר גמיש, תחביר מעט יותר מסורבל

# 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) ... {
    VersiondString other_vs;
    ...
    if (this.n > other_vs.length())
      return  1;
```

- יותר יעיל, התייחסות לשדה מופע n במקום קריאה לשירות

# 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 החוזה
- למשל, החוזה של האיטרטור ש-CS מחזירה מבטיח שהאיברים יוחזרו בסדר עולה, מכיוון שהמערך שמכיל אותם ממוין

# טיפוס סטאטי 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?

# שיגור דינמי dynamic dispatch

- 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 DT שיורש מ ST . זהו הטיפוס הדינמי של `target`
- 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)
- הקומפיילר יוצר קוד לביצוע הבחירה בזמן קבוע

# חיזוק החוזה

- בדוגמה רואים שימוש לחיזוק החוזה: אם לקוח צריך עצם ממחלקה עם חוזה משופר (CS), הוא צריך להשתמש במשתנה מטיפוס סטאטי CS שמתייחס לעצם עם טיפוס דינאמי CS
- זה נותן ללקוח אפשרות בחירה: חוזה רגיל דרך C, חוזה משופר דרך CS
- למה שהלקוח יבחר בחוזה חלש יותר? אולי המימוש של החוזה החלש 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

# 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 שהוספנו 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 `i` עד `length()`
- מבחינה מסוימת, חיזקנו את החוזה, כי הוספנו שירות שלא היה קודם, אבל מבחינה אחרת, החלשנו את החוזה (את הספק), כי לספק החדש יש שירות עם תנאי קדם יותר מגביל
- רעיון מפתה, אבל לא מוסרי

# עקרון ההחלפה (substitution)

- לקוח שמשתמש 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

# ולכן אסור להרחבה להחליש 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 (נסביר בהמשך; 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 את תנאי הקדם (תנאי אחר נוסף `true`) או רק תנאי האחר (תנאי קדם נוסף `false`)

# 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, then chop the old ones
doIt(bvs);
```

תקין תחבירית אבל מפר את עקרון ההחלפה

# 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
- ולמנשקים אין בנאים כלל (למחלקה מופשטת יש בנאי, לצורכי המחלקות היורשות).

# 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

# 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
```

# 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

# שבריות

```
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 הקומפיילר לא יגלה את זה אם זה בקובץ אחר, והתוכנית תמשיך לעבוד, ולקרוא לגרסה הירוקה
- לא טוב שקומפילציה רק של קובץ שלא השתנה תשנה את התנהגות התוכנית; זה מצב שבריר; פגם בג׳אווה

# אולי יותר גרוע

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

# מה ייבחר

- מנגנון ההעמסה הוא סטטי: בוחר את החתימה של השרות (טיפוס העצם, שם השרות, מספר וסוג הפרמטרים), אבל עדיין 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 )
```

# 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 שבריריות, 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

# 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 שייוצר יחס בין הטיפוסים

# 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 בהיררכית 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 מממשות את אותו מנשק

# 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

# 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 המשתמר של המורחבת חייב להתקיים אחרי סיום פעולת שירות של מחלקת הבסיס
- כלומר, המשתמר של המחלקה המרחיבה יכול להיות חזק יותר, אך לא חלש יותר

# 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 דרוסים של מחלקת הבסיס לא יפרו אותו
- איך דואגים לכל זה?

# 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

# 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

# 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();
  } }
```

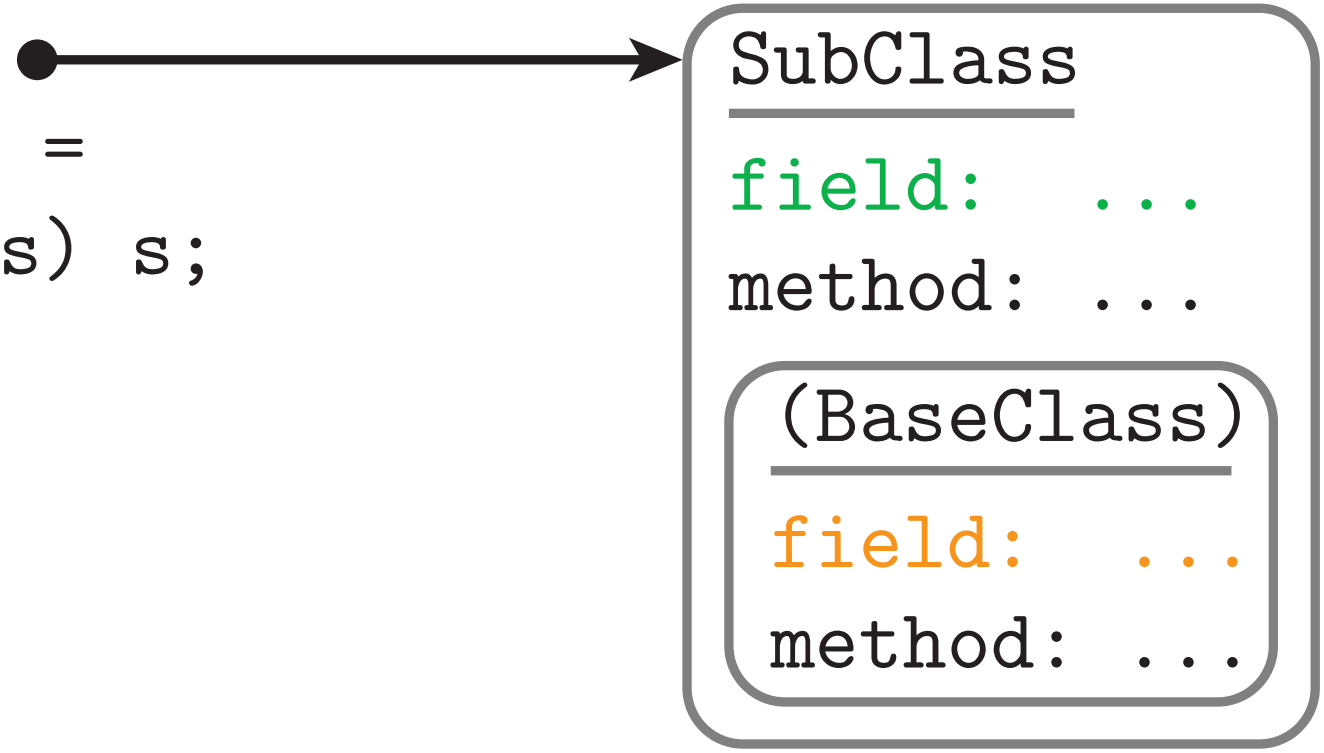
- השירות הדורס מחזק את החוזה של הנדרס על ידי הוספת תוצא לוואי לנדרס; דוגמאות אחרות מחזקות את תנאי האחר
- השירות הדורס אחראי לקיום תנאי הקדם של הנדרס ולקיום המשתמר של מחלקת הבסיס בנקודה שבה הנדרס נקרא

# 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 מהתייחסות לשירותים: אליהם מתייחסים תמיד לפי הטיפוס הדינמי
- ניתן לבחור את שדה המופע על ידי `super.f` בשירותים של `S` או על ידי יציקה `((B) this).f`

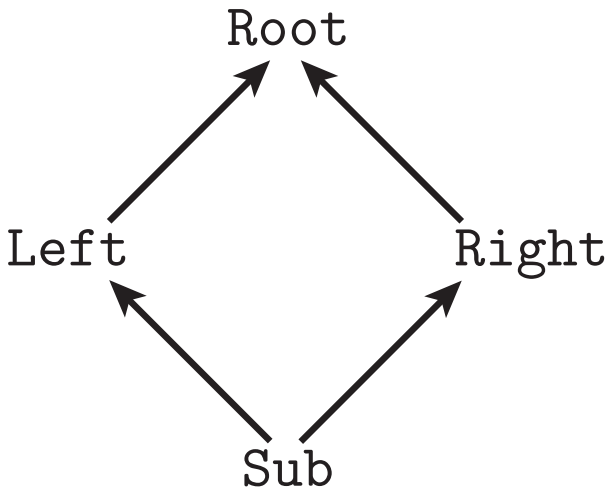
# התייחסות סטאטית לשדות מופע

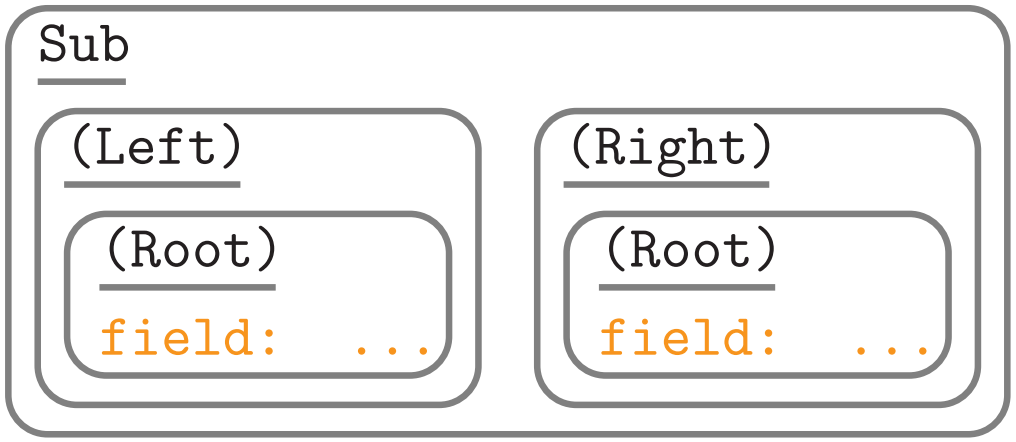
```
SubClass s:
BaseClass b =
   (BaseClass) s;
s.method()
b.method()
s.field
b.field
```

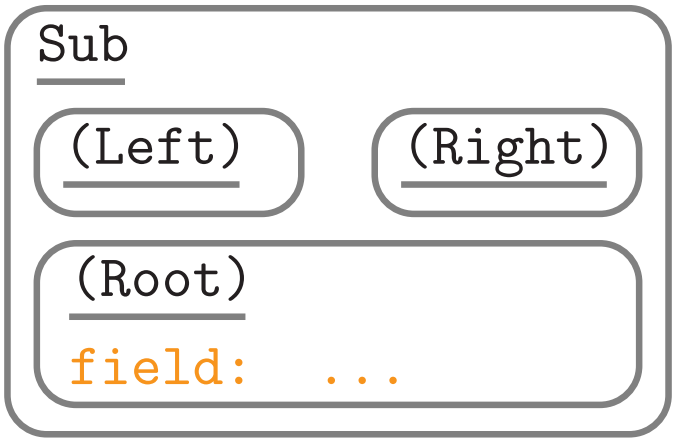


# זה מעורר עוד סוגיה בירושה מרובה

- האם לשתף אב קדמון או לא?



# 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
- המימוש של Sub צריך לבחור האם לשתף; העסק מסתבך

# מה עוד לא יורשים

- גם שרותי מחלקה (`static`) מוסתרים ולא נדרסים (ואם נקפיד 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
- לא יורשים שדה או שרות פרטי; אם מופיע שדה או שרות באותו שם במחלקה היורשת, אין לו כל קשר לזה שבמחלקת הבסיס

# 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 משימוש בה
- 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 וזה עלול להיות נורא

# 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)

# 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`
- וזה מבטיח שכל עצם בג׳אווה מספק שירותים בסיסיים מסוימים, אם על ידי ירושה מ-`Object` או על ידי דריסה
- השירותים הללו הם `equals`, `clone`, `toString`, ועוד כמה פחות חשובים

# תבנית נוספת: call back

- שרות מוחשי במחלקה מופשטת יכול לקרוא לשירות מופשט
- מנגנון של call back (קוד "מערכת" מפעיל קוד משתמש)

```
abstract class Stack {
  public void change_top(String t) {
     pop(); push(t);
  }
  abstract public void push(String t);
  abstract public void pop();
}
```

# 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
}
```

# סיכום הרחבה וירושה

- מאפשרת להוסיף למחלקה שדות מופע ושירותים או לדרוס שירותים ולהגדיר אותם מחדש
- מאפשרת שימוש נוסף בקוד ללא שכפול, הרחבה ותיקון של מחלקות שאין את קוד המקור שלהן
- למחלקה מרחיבה או מחלקה שממשת מנשק אבל משנה את החוזה שלו אסור להחליש את החוזה, אבל שפת התכנות לא כופה את האילוץ 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