

חלק 2

עצמים ומחלקות

# למה תכנות מונחה עצמים?

- בהנדסה קורות לעיתים קטסטרופות: בניינים קורסים, מטוסים נופלים, כורים מתפוצצים
- מקטסטרופות לומדים
- בעולם המחשבים, רוב הקטסטרופות התבטאו בכישלון לפתח תוכנה גדולה או בכישלון להשמיש תוכנה שפותחה; רוב הקטסטרופות נבעו מהגודל של התוכנה
- הפקת הלקחים כללה את פיתוח המתודולוגיות של תכנות מונחה עצמים, תיכון בעזרת חוזים (design by contract), ביצוע מקסימום בדיקות תקינות בזמן קומפילציה, ניהול זיכרון אוטומטי

# מודולריות

- מודולריות היא תכונה חשובה של תוכנה.
- נחוצה כדי לאפשר הפרדת עניינים בזמן הפיתוח, ולשפר קריאות לצורך תחזוקה.
- מודולריות פירושה היכולת לפרק מערכת למרכיבים, לבנות מערכת ממרכיבים, להבין כל מודול בפני עצמו, רציפות, הגנה
- מודולריות טובה כתכונה של מערכת דורשת מודולים בעלי חוזק פנימי גבוה, וצמידות נמוכה
- מתברר שארכיטקטורת מערכת שמבוססת על הנתונים מאפשרת מודולריות טובה יותר מארכיטקטורה שמבוססת על הפונקציונליות
- מכאן היתרון של פיתוח תוכנה מונחה עצמים

# שימוש חוזר בתוכנה

- על מנת לשמור על עלויות תוכנה סבירות, יש לשפר את תפוקת מפתחי התוכנה

- שיפור תפוקה יומית של מתכנת דורש שיפורים משמעותיים בתהליכי הפיתוח, שפות התכנות, וכלי הפיתוח

- בנוסף, ניתן להקטין את עלות הפיתוח ע"י שימוש ברכיבי תוכנה קיימים, שפותחו עבור פרויקט קודם או פותחו במיוחד כתשתית לארגון

- שימוש חוזר בתוכנה כרוך בקשיים רבים, לא כולם טכניים: תסמונת "לא הומצא אצלנו", תשלום עבור תוכנה לפי שורות

קוד

- הניסיון מראה שרכיבי תוכנה מונחת עצמים מתאימים לשימוש חוזר יותר מרכיבים פרוצדורליים

# עצמים ומחלקות

- עצם (object) הוא יחידת תוכנה שמספקת שירותים (methods) מסוימים ושיש לה בכל נקודת זמן מצב רגעי מסוים (state)
- מחלקה (class) היא קבוצה של עצמים מאותו סוג, כלומר שמספקים את אותם שירותים באותה צורה
- העצמים הם מופעים (instances) של המחלקה
- עצמים שונים מאותה מחלקה נמצאים במצבים רגעיים שונים
- המחלקה היא הישות הסטטית בקוד המקור; העצם הוא הישות הדינמית בזמן הריצה
- ב scheme מימשנו גירסא פשוטה של מחלקות ועצמים ע"י פרוצדורות עם משלוח הודעות.

# שירותים לעומת פרוצדורות

- קופת קולנוע היא עצם שמספק שירות: מכירת כרטיסים
- השירות שלקוח מקבל תלוי במצב הרגעי של העצם: כמה כרטיסים כבר נמכרו ואיזה
- מספרה היא פרוצדורה: הלקוח נכנס ויוצא מסופר בלי קשר למצב של המספרה או לשירות שקיבלו לקוחות קודמים
- (הדוגמאות הללו מתעלמות מתור בקופת הקולנוע או במספרה, תור שמהווה סוג של מצב נוכחי. הדוגמאות מניחות שכאשר הלקוח שלנו מגיע, אין תור. התור גם לא משפיע על התוצאה הסופית עבור הלקוח, רק על הזמן שדרוש על מנת לקבל את השירות.)

# טיפוסים (Types)

- ב-scheme כל ערך שייך לטיפוס נתונים מסוים, אבל משתנה יכול להכיל ערך מטיפוס כלשהו ללא מגבלה
- כלומר טיפוס הוא תכונה דינמית (משתנית עם הזמן במהלך ביצוע התכנית)
- בג'אווה וברוב השפות האחרות הטיפוס הוא תכונה סטטית:
- כאשר מגדירים משתנה, קובעים מה יהיה הטיפוס שלו
- בזמן ריצה ערכו של המשתנה יכול להשתנות, אבל הטיפוס יישאר ללא שינוי

# טיפוסים

- בשפות מונחות עצמים (למשל, Python, C++, Java, C#, Smalltalk) מחלקות הן גם טיפוסים
- לכל עצם בזמן הריצה יש טיפוס: המחלקה שאליה הוא שייך
- בשפות שבודקות טיפוסים בצורה סטטית (בזמן קומפילציה), משתנים, שמות בתוכנית, מוכרזים עם טיפוס: אם משתנה מתייחס למשהו בכלל, המשהו הזה הוא עצם מהמחלקה המוכרזת
- לחוקק שתארונו יש יוצאים מן הכלל שנלמד בהמשך
- ג'אווה בודקת טיפוסים בצורה סטטית, וכמוה גם C++, Python, Smalltalk לא בודקות (בדומה ל scheme)



# מחלקות וטיפוסים: דוגמה

נגדיר מחלקה

```
class VersionedString {...}
```

במקום אחר בתוכנית, נגדיר משתנה עם טיפוס מתאים, ומשתנה מטיפוס `String` (מחלקה קיימת שאין צורך להגדיר):  
`VersionedString vs;` המשתנה עדיין לא מתייחס לעצם

`String s;` כנ"ל

ניצור עצם חדש מהמחלקה, ונקשור את המשתנה `vs` אליו,  
`vs = new VersionedString();`

אבל אי אפשר לקשור שם מטיפוס `String` לעצם מהמחלקה  
`:VersionedString`

`s = vs;` שגיאת קומפילציה!

# מחלקה ראשונה: מחרוזת עם היסטוריה

כעת נגדיר מחלקה. ראשית, נגדיר במילים מה עצמים מהמחלקה ייצגו ואיזה שירותים הם יספקו:

- עצם מייצג סדרה של גרסאות של מחרוזת
- השירותים שהעצם יספק הם הוספת גרסה עדכנית למחרוזת, שליפת הגרסה העדכנית (אחרונה), שליפת גרסה ישנה מסוימת, וספירת מספר הגרסאות של מחרוזת
- לא עצם שימושי כל כך, אבל עצמים דומים שמייצגים סדרת גרסאות של קובץ הם כן שימושיים
- שימוש במחרוזות במקום קבצים מפשט את ההדגמה

# המחלקה הראשונה: הגדרת השירותים

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

- שיטות ציבורי, אין הגבלת גישה: `public`
- מאומה; ערך חזרה שמסמן שהשיטות אינו מחזיר ערך: `void`
- מספר שלם: `int`
- מחלקה לייצוג מחרוזות, מובנית בשפת ג'אווה: `String`

# מה השירותים עושים? מצב מופשט

- הדרך הנוחה ביותר להגדיר מה השירותים עושים (ולחזותם) שהם עושים זאת נכון) היא על ידי הגדרת המצב מופשט (abstract state) שהעצם מייצג
- בעיני הלקוח, עצמים מייצגים מצבים מופשטים
- המצב המופשט של עצם מהמחלקה `VersionedString` הוא סדרה  $(s_1, s_2, \dots, s_n)$  כאשר  $n \geq 0$  ו- $s_i$  היא מחרוזת
- את המצב המופשט של העצם נסמן ב- $A(\text{this})$

# מה השירותים עושים? החזרה

class VersionedString:

*Initial State:  $A(this) == ()$*

add(String s) :

*Requires:  $s \neq \text{null}$*

*Ensures:  $A(\text{old this}) == (s_1, s_2, \dots, s_n)$   
 $\Rightarrow A(\text{this}) == (s_1, s_2, \dots, s_n, s)$*

# מה השירותים עושים? החזרה (המסך)

`int length()` :

*Requires: nothing*

*Ensures :  $A(this) == (s_1, s_2, \dots, s_n)$*

$\Rightarrow return == n$

`String getVersion(int i)` :

*Requires:  $0 < i \leq length()$*

*Ensures :  $A(this) == (s_1, s_2, \dots, s_n)$*

$\Rightarrow return == s_i$

# החזרה (המשך)

String getLastVersion() :

*Requires: length() > 0*

*Ensures: A(this) == (s<sub>1</sub>, s<sub>2</sub>, ..., s<sub>n</sub>)*

$\Rightarrow return == s_n$

- הסימונים: Requires תנאי קדם, Ensures תנאי אָחֵר, old הערך לפני ביצוע השרות, return הערך שהשרות מחזיר
- השרות add משנה את המצב המופשט (פקודה), האחרים לא (שאליות)

# החזרה: תנאי קדם ותנאי אחר

- לעצמים יש מצב התחלתי
- לכל שירות מוצמדים שני תנאים
- תנאי הקדם (precondition) מגדיר מה השירות מצפה
- תנאי האחר (postcondition) מגדיר מה השירות מספק
- אם תנאי הקדם מתקיים, השירות חייב לקיים, לאחר שהוא מסיים, את תנאי האחר
- אם תנאי הקדם לא מתקיים, השירות לא מחויב לכולם; לא לעצור, לא להימנע מלהעריך את התוכנית, לא להימנע מפגיעה במבני נתונים, כלום



# ספקים ולקוחות

- לחוזה שני צדדים: ספק ולקוח
- הספק הוא המחלקה שמגדירים; היא צריכה לממש את השירותים בקוד ג'אוה מתאים
- הלקוח הוא קוד שמתממש בעצמים מהמחלקה
- הלקוח מחויב לקיים את תנאי הקדם לפני שהוא קורא לשירות
- הספק מחויב, אם הלקוח קיים את חלקו ותנאי הקדם מתקיים, לקיים את תנאי האחר

# החזרה בלי הגדרת מצב מופשט

בהרבה מקרים אפשר להגדיר את החזרה תוך שימוש בשאילתות בלבד, בלי להגדיר את המצב המופשט כלל; זה אפשרי כאשר השאילתות חושפות את כל המצב המופשט; לפעמים זה מקשה על הגדרת החזרה והוכחת הנכונות; (פריט שלא מופיע בתנאי האחר לא השתנה); הנה הדוגמה

```
class VersionedString:
```

```
Initial State: length() == 0
```

```
add(String s) :
```

```
Requires: s != null
```

```
Ensures: length() == old length()+1
```

```
getVersion(length()) == s
```

# החזרה בלי הגדרת מצב מופשט (המשך)

`int length()` :

*Requires: nothing*

*Ensures : return == number of calls to add() so far*

`String getVersion(int i)` :

*Requires:  $0 < i \leq \text{length}()$*

*Ensures : return != null*

`String getLastVersion()` :

*Requires:  $\text{length}() > 0$*

*Ensures : return == getVersion(length())*

# שימושי החזרה: מה רואה הלקוח

- הקוד של המחלקה אינו מתפרסם (אינו ידוע ללקוחות)
- רק החזרה מתפרסם
- לקוח שמתמש במחלקה מסתמך על החזרה שלה
- הלקוח יכול לעקוב אחרי פעולת המחלקה באמצעות החזרה, ויכול לאמת את הקוד שלו שמתמש במחלקה
- באופן כזה נעשית חלוקת אחריות בין כותב המחלקה ללקוח של המחלקה
- כותב המחלקה אחראי להבטיח שהמחלקה מקיימת את החזרה
- הלקוח אחראי לכך שהוא מפעיל את המחלקה בהתאם לחזרה

# איך נבדוק נכונות של לקוח?

נניח שהלקוח מבצע את סידרת הפעולות הבאה:

```
vs = new VersionedString();  
vs.add("The letter A");  
vs.add("The letter B");  
System.out.println(vs.getVersion(1));
```

איך ניתן להראות שהפעולות ייתבצעו בהצלחה, ומה יודפס?

● (השגרה `println` מ `System.out` מדפיסה את הארגומנט שלה, שצריך להיות מחרוזת, לפלט הסטנדרטי, ועוברת לשורה הבאה. בהמשך הקורס נלמד מה משמעות שם השגרה)

# נכונות של לקוח

ליצירת העצם אין תנאי־קדם, ולכן מותר ללקוח לבצע אותה כעת (או בכל מצב אחר)

```
vs = new VersionedString();
```

לאחריה מתקיים:

```
vs.length() == 0
```

לשירות add יש תנאי קדם אחד: הארגומנט אינו null. הלקוח העביר התייחסות למחרוזת, לא null, ולכן מילא את תנאי הקדם.

```
vs.add("The letter A"); argument != null
```

תנאי האחר מבטיחים ש-`length()` קודם ב-1 ושריקיאה ל-`getVersion(1)` תחזיר את המחרוזת שהועברה, כלומר מתקיים:

```
vs.length() == 1
```

```
vs.getVersion(1) == "The letter A"
```

## נכונות של לקוח (המשך)

באופן דומה

```
vs.add("The Letter B"); argument != null  
vs.length() == 2, vs.getVersion(2) == "The letter B"
```

עקרונית, יתכן שפקודה מאוחרת יותר תשנה את הערך שייחזיר

(2) `getVersion()` במחלקה שלנו זה לא יתכן, כי הערך של `length()` יכול רק לגדול, והפקודה היחידה היא `add`, שקובעת את הערך של `getVersion()` עבור הארגומנט `length()`

עכשו מתקיים תנאי הקדם של `vs.getVersion(1)`

```
0 < 1 <= vs.length() == 2
```

```
System.out.println(vs.getVersion(1));
```

ויודפס "The letter A"

## **למה חוזים?**

- החוזה מגדיר את המשמעות של מחלקה ללא תלות במימושה
- החוזה מאפשר להפריד את הפיתוח והתחזוקה של הספק מאלו של הלקוחות; ההפרדה הזו מהווה מפתח בפיתוח תוכנה רחבת היקף
- חוזה פורמלי מאפשר להוכיח נכונות של לקוחות
- בהמשך הקורס נראה שהחוזה הוא מרכיב (לא בלעדי) בהוכחת נכונות של ספק
- בהמשך הקורס נראה גם שהגדרת המשמעות של מחלקה על ידי חוזה חשובה במיוחד בתכנות מונחה עצמים
- כמובן שיש חוזה טוב וחוזה פחות טוב; בהמשך הקורס נציע שיטות להגדרת חוזים טובים



# חוזים טובים לעומת חוזים פחות טובים

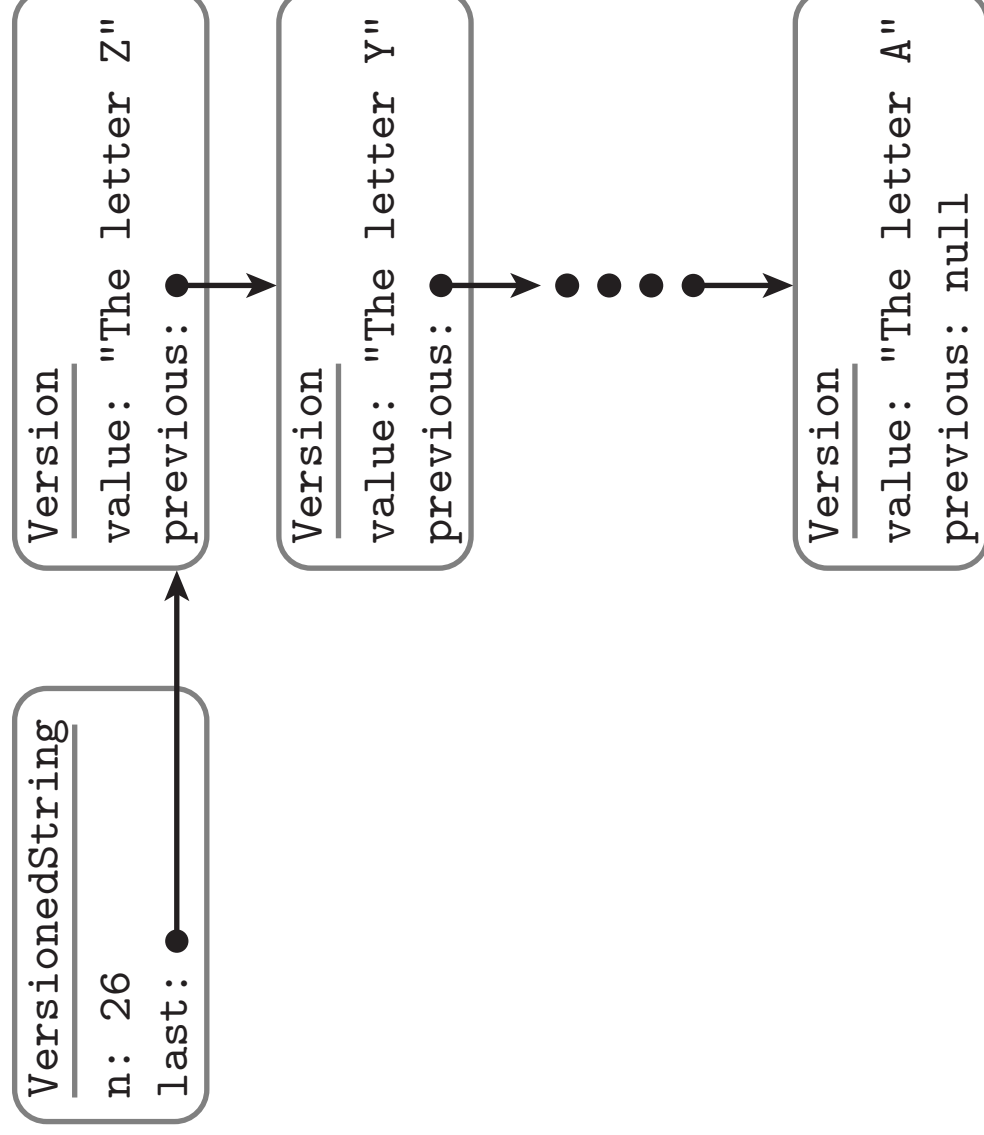
- הגדרת חוזים מוצלחים היא נקודה חשובה ומורכבת בתיכון תוכנה, ובהמשך הקורס נדון בה בפרוטרוט. אבל כדאי כבר עכשיו להתחיל לחשוב על מה הופך חוזה לטוב או לגרוע. אפשר לחשוב על כך בהקשר של חוזים בעולם הממשי, לא דווקא בהקשר של חוזים בין מחלקות.

- חוזה טוב הוא חוזה שקל להבין אותו, שאפשר לצפות ששני הצדדים יוכלו לעמוד בו, ושבימדת האפשר, כל צד יכול לוודא את עמידת הצד השני במילוי התחייבויותיו

# פקודות ושאליות

- השירות add במחלקה שהגדרנו הוא פקודה (command):  
הוא משנה את מצב מבנה הנתונים
- השירותים `getVersion`, `length` הם שאליות (queries): הם מחזירים מידע אודות מצב מבנה הנתונים, אבל לא משנים אותו
- הפרדנו בין פקודות ושאליות: אין שירותים שהם גם פקודה וגם שאלית
- חשיבות ההפרדה: מקילה על הבנת הממשק של מחלקה, מקילה על הגדרת החוזה: מאפשרת שימוש בשאליתא בחוזה לעיתים (רחוקות) יש סיבות טובות לא להפריד (ביצועים,...)
- בהיעדר סיבה טובה, הפרידו!

# מימוש המחלקה: הרעיון



## מימוש המחלקה: השדות

המצב הרגעי של עצם נשמר בשדות, משתנים ששייכים לעצם:

```
class Version {
```

```
    String value;          הערך של גרסה זו
```

```
    Version previous;     התייחסות לגרסה הקודמת, אם יש  
}
```

```
class VersionedString {
```

```
    protected int        n;          מספר הגרסאות
```

```
    protected Version last;         התייחסות לגרסה אחרונה
```

```
    ...
```

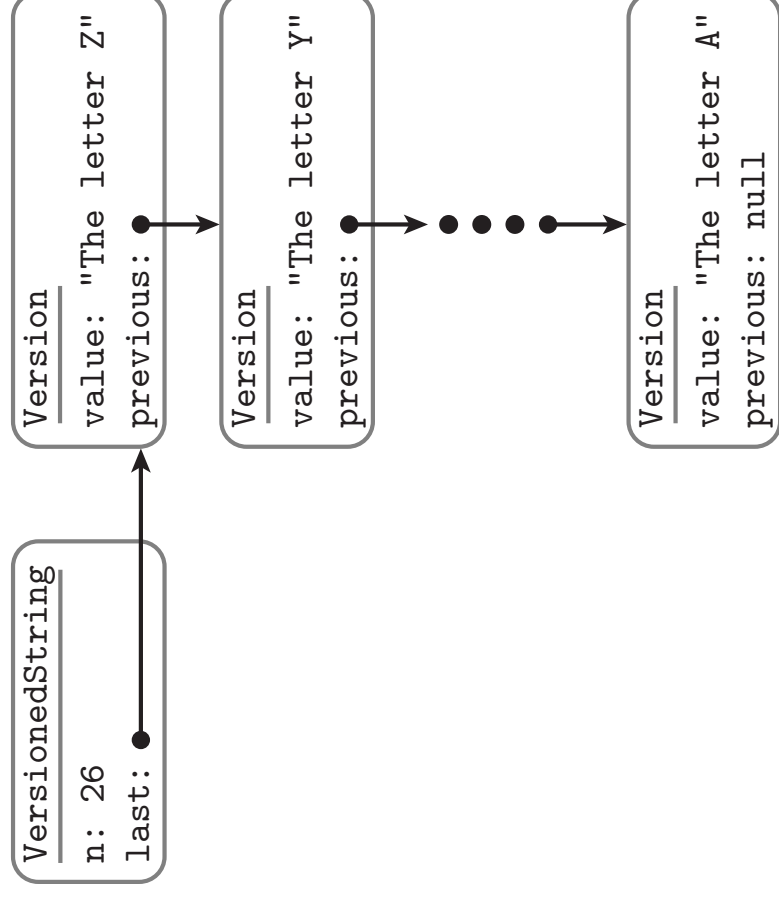
```
}
```

# שדות (fields) של עצם

- השדות של עצם הם קבוצה של משתנים. הערכים שלהם מייצגים את המצב הרגעי של העצם. לכל עצם ממחלקה מסוימת יש שדות פרטיים לו
- כלומר המחלקה היא מעין תבנית של משתנים; כאשר יוצרים עצם מהמחלקה, נוצק מהתבנית הזו עצם שיש לו עותק פרטי של כל אחד מהשדות
- השדות של עצם מאותחלים באופן אוטומטי כאשר העצם נוצר. את חוקי האתחול נלמד בהמשך

# מימוש המחלקה: הפקודה

```
public void add(String s) {  
    Version l = new Version();  
    l.previous = last;  
    l.value = s;  
    last = l;  
    n = n+1;  
}
```



## מימוש המחלקה: השאיליות

```
public String getLastVersion() {  
    return getVersion( length() );  
}  
  
public String getVersion(i) {  
    Version v = last;  
    for (int j = length(); j > i; j--)  
        v = v.previous;  
    return v.value;  
}  
  
public int length() { return n; }
```

# פיתוח תוכנה מונחית עצמים

- מערכת תוכנה מדמה עולם מציאותי מסוים
- העולם מורכב מישויות שלכל אחת מהן ידע מסוים, ויכולת לבצע פעולות, או לספק שירותים; אלה העצמים
- בפיתוח המערכת יש לזהות מהן הישויות המרכיבות אותה (עצמים), ולסווג אותם למחלקות
- לכל מחלקה, צריך לקבוע מה "יודע" עצם מהמחלקה, ומה השירותים שהוא מספק



# מודל הביצוע של תכנית מונחית עצמים

- בזמן ביצוע התכנית קיימים בזיכרון מספר עצמים, שחלקם מתייחסים זה לזה
- בכל רגע נתון, מתבצעת פעולה מסוימת בעצם אחד
- כאשר עצם  $X$  מבצע פעולה מסוימת, הוא יכול לבקש מעצם אחר  $Y$  (שיש לו התייחסות אליו) שרות מסוים
- $X$  שולח ל  $Y$  הודעה וממתין עד ש  $Y$  יסיים את הפעולה (ויחזיר ערך), ואז  $X$  ממשיך בפעולתו
- (בג'אווה ובשפות דומות ניתן להפעיל מספר "מעבדים וירטואליים", threads, בו זמנית על קבוצת העצמים שבזיכרון; זה דורש תיאום בין ה-threads; כרגע לא נדון בכך)

# השמה של ערכים פרימיטיביים

השמה מעתיקה ערך פרימיטיבי ממשתנה אחד לאחר

```
int x = 5;  
int y = x;  
y = 3;
```

הערך של  $x$  עדיין 5, משום שהערך שלו רק הועתק למשתנה  $y$   
ששונה בהמשך

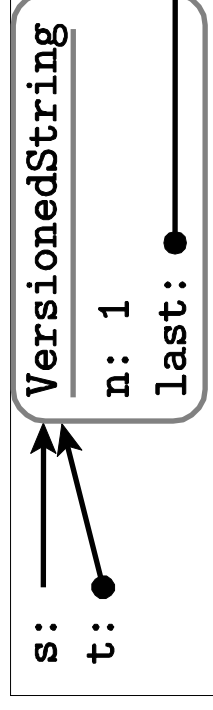
- משתנה פרימיטיבי הוא מיכל לאחסון ערך
- לא כמו נוזל, שמוזגים ממיכל למיכל!

# השמה של התייחסויות

השמת התייחסות מעתיקה את ההתייחסות, לא את העצם שמתייחסים אליו:

```
VersionedString s = new VersionedString();  
s.add("V 1");  
VersionedString t = s;
```

אחרי ההשמה, שני המשתנים מתייחסים לאותו עצם בדיוק:



```
שינוי מצב העצם דרך t משנה את העצם ש-s מתייחס אליו:  
t.add("V 2");  
s.getLastVersion(); // returns "V 2"
```

# העברת ארגומנטים

- כאשר מתבצעת קריאה לשרות, ערכי הארגומנטים נקשרים לפרמטרים הפורמליים של השרות לפי הסדר, ומתבצעת השמה לפני ביצוע גוף השרות.

- בהעברת ערך פרימיטיבי הערך מועתק לפרמטר הפורמלי:

```
void f(int y) { y = 3; }
```

```
int x = 5;
```

```
f(x); // x still contain 5; equivalent to { y=x; y=3; }
```

- העברת התייחסות כארגומנט מעתיקה את ההתייחסות, לא את העצם שמתייחסים אליו

- צורה זאת של העברת פרמטרים נקראת `call by value`

## למה יש מערכים בג'אווה?

- במקום מערך ניתן להשתמש בג'אווה בעצמים. למשל, במקום מערך של doubles ניתן להשתמש במחלקה

```
class DoubleArray {  
    public void put(int index, double value) {.}  
    public double get(int index) {...}  
    ... // משתני מופע  
}
```

- מדוע, אם כך, כוללת השפה מערכים? התשובה היא יעילות, מבחינת זמן ריצה וזיכרון; לעיתים יעילות חשובה, לפעמים לא
- מערכים בג'אווה הם פשרה: יותר יעיל, פחות אלגנטי