

# חלק 5

## מְנַשְׁקִים

# (לפני שנתחיל, ובלי קשר לנושא)

## שדות מחלקה ושרותי מחלקה

- השדות והשירותים שדברנו עליהם עד כה נקראים שדות מופע ושרותי מופע - הם מתייחסים לכל מופע (עצם):
- לכל עצם עותק משלו של כל שדות המופע
- כל שרות פועל על עצם, וניגש (לקריאה או כתיבה) לשדות של העצם הנוכחי `this`
- יש גם סוג אחר של שדות ושירותים, שמוגדרים עם המילה `static` לפניהם (כ `modifier` בלי קשר אם הם `private` או `public` וכו'). הם נקראים שדות מחלקה ושרותי מחלקה

# שדות מחלקה ושרותי מחלקה (המשך)

- לשדה מחלקה יש עותק אחד, ולא עותק לכל עצם.
- שימוש לדוגמא: מספר העצמים ממחלקה זאת שנוצרו.
- שימוש נפוץ נוסף: קבועים גלובליים (final - השמה אחת)

```
public static final double PI = 3.14159
```

- שרותי מחלקה אינם פועלים על עצם מסוים, ולכן אינם יכולים לגשת לשדות מופע, אלא רק לשדות מחלקה.

- שימושים נוספים: כאשר אין התייחסות לעצם, כגון פונקציות מתמטיות, וכמובן ה main

- הגישה היא באמצעות שם המחלקה, לא שם העצם. דוגמא

```
Math.sin(...)
```

# שדות מחלקה ושרותי מחלקה (דוגמה)

```
public class SomeClass {  
    private static int count = 0;  
    private AnyType x ..... // can be public etc.  
    public SomeClass () {  
        count++; // additional code ....  
    }  
    public static int getCount () {  
        // cannot access x here!!  
        return count;  
    }  
}
```

# הוראות לניקוי אקוואריום

How to clean your aquarium safely, by Sarah Davies

...

Turn off and remove all heaters and filters. These can be put in the sink and left until they are cleaned. Fill one of the new, clean buckets half full of water from the aquarium. Using the fish net, transfer **the fish**, one by one, to this bucket until all the fish are out of the aquarium. Next, remove all plants and ornaments. If the plants are living put them in the bucket with the fish. Put all **the ornaments** on the counter or in **the bucket** where the rocks go....

# **איזה דגים, קישוטים, ודליים בדיוק?**

- כמובן שזה לא משנה; אותן ההוראות תקפות לאקוואריום עם דגי זהב ולאקוואריום עם ברקודות וכרישים, לדלי פח כחול ולדלי פלסטיק אדום
- ההוראות מתייחסות לעצמים באופן כללי שמאפשר שימוש בהוראות בהרבה מצבים שונים
- החוזה של הדלי: נוזלים ומוצקים שמכניסים לתוכו נשאים שם אם לא ממלאים אותו יותר מדי, מה שנכנס אפשר להוציא בדיוק באותו מצב (בפרט אסור שיהיו בדלי שיירי סבון)
- המימוש יכול להיות מפלסטיק, מפח, ואפילו מימושים כמו אגרטל חרסינה או סיר נירוסטה ממלאים את הדרישות

# הקוד שלנו, עד עתה, לא היה כל כך כללי

- הלקוח שהשתמש בעצם מהמחלקה `VersionedString` עשה זאת דרך ייחוס מטיפוס `VersionedString`

- עצם מהמחלקה הזו מקיים כמובן את החוזה שהלקוח מסתמך עליו, אבל אולי יש עוד הרבה מחלקות שמקיימות את החוזה הזה

- למה שקוד הלקוח לא יוכל לפעול על כל מחלקה כזו?, למשל,

```
int Find(VersionedString vs, String s) {  
    for (int i=0; i<vs.length(); i++)  
        if (s.equals( vs.getVersion(i) ))  
            return i;  
}
```

# זה פועל, אבל

- זה לא מאפשר להשתמש בשגרה הזו, המימוש הזה של אלגוריתם חיפוש, במצבים אחרים
- זה מקביל ל-"קח את דלי הפלסטיק האדום שמתחת לכיור (לא את דלי הספונג'ה הכחול), והעבר אליו את המים ואת דג הזהב"
- הפתרון: להפריד את הספק, המחלקה שממשת את השירותים, מהחווה, שאינו תלוי במימוש



# מְנַשְׁקִים (interfaces)

המנשק מגדיר את המנשקים של השירותים ואת החוזה

```
interface VersionedString {  
    public void    add(String s)        ;  
    requires ... ; ensures: ...  
    public int    length()              ;  
    requires ... ; ensures: ...  
    public String getLastVersion()      ;  
    requires ... ; ensures: ...  
    public String getVersion(int i)    ;  
    requires ... ; ensures: ...
```

## מְנַשֵּׁקִים (המשך)

- מנשק אינו מכיל מימוש כלשהו. השירותים מופיעים ללא גוף, רק כותרת שלאחריה ; (וחוזה).
- כל השירותים שמופיעים מיוצאים - למעשה ניתן להשמיט את ה public לפניהם. (כתיבת private למשל היא שגיאה).
- במנשק לא יכולים להופיע שדות (כי הם חלק ממימוש)
- במנשק לא יכולים להופיע שירותי מחלקה.
- לא ניתן ליצור עצמים מן המנשק, ולכן לא יכולים להופיע בנאים.

# ספק מממש מנשק

ספק שמממש מנשק מקיים את החוזה שלו; אין לו חוזה משלו

```
class LinkedVersionedString
    implements VersionedString {
    protected int      n;
    protected Version  last;
    public void      add(String s)      {...}
    public int       length()           {...}
    public String    getLastVersion()   {...}
    public String    getVersion(int i) {...}
}
```

# הספק והחזקה

- הספק חייב לקיים את החזקה של המנשק שהוא מממש. ככלל, אין לו חזקה משלו
- הספק חייב לספק שירות אם המצב התוכנית מקיים את תנאי הקדם של השירות
- אם מצב התוכנית מקיים את תנאי הקדם של שירות, אזי מצבה לאחר סיום השירות חייב לקיים את תנאי האחר
- אולי השירות יפעל גם כשלא מקיימים את תנאי הקדם, ואולי השירות מביא למצב טוב מזה הנדרש בתנאי האחר
- אבל שירות טוב יותר מזה המובטח בחזקה אינו נדרש, וגם אם הספק מכריז על החזקה המשופר שלו, עדיף אולי ללקוח להימנע מלהסתמך עליו, כי אחרת לא יוכל להחליף ספק

## **דוגמא: מחסנית**

- נגדיר מנשק למחסנית, בדומה למחלקה שראינו
- כמובן בלי אף רמז למימוש
- נוסיף גם שאילתה `full()` לבדוק אם המחסנית מלאה (כי במימושים מסוימים יתכן מצב כזה).
- המנשק כולל את החוזה (עם שינויים שנובעים מהוספת האפשרות שהמחסנית מלאה).

```
/**
 * The Stack interface represents ....
 * @param <T>
 */
interface Stack <T>{
    /**
     * @pre !empty()
     * @return the top element
     */
    public T top ();
}
```

```
/**
 * Add an element @param t to top of stack
 * @pre !full()
 * @post !empty()
 * @post top() == t
 */
public void push(T t);
/**
 * @pre !empty()
 * @post !full()
 */
public void pop();
```

```
/**
 * @return true if the stack is empty
 */
public boolean empty();
/**
 * @return true if the stack is full
 */
public boolean full();
}
```



# מחלקות למימוש המנשק

- נגדיר שתי מחלקות למימוש המחסנית
- המחלקות יורשות את החוזה מהמנשק
- מימוש מקושר `LinkedStack <T>` (דומה למימוש שראינו)
- השאילתה `full()` מחזירה תמיד `false`
- מימוש בעזרת מערך `LinkedStack <T>`
- יצירת המערך דורשת ביטוי שיוסבר בעתיד

```
/**
 * The LinkedStack class is a linked
 * implementation of Stack the interface.
 * Initially the Stack is empty.
 * @param <T>
 */
public class LinkedStack <T>
    implements Stack <T> {
    private Cell <T> top;
```

```
public T top () {
    return top.cont();
}

public void push(T t) {
    Cell <T> x = new Cell <T> (t,top);
    top = x;
}

public void pop() {
    top = top.next();
}
```

```
public boolean empty() {  
    return (top == null);  
}
```

```
public boolean full() {  
    return false;  
}  
}
```

```
/**
 * The FixedCapacityStack class implements
 * the Stack interface using an array
 * Capacity is the constructor parameter
 */
public class FixedCapacityStack <T>
    implements Stack <T> {
    private T [] content;
    private int capacity;
    private int topIndex;
```

```
public FixedCapacityStack (int capacity) {  
    content = (T[]) new Object[capacity];  
    this.capacity = capacity;  
    topIndex = -1;  
}  
  
public T top () {  
    return content[topIndex];  
}
```

```
public void push(T t) {  
    content[++topIndex] = t;  
}
```

```
public void pop() {  
    topIndex--;  
}
```

```
public boolean empty() {  
    return (topIndex < 0);  
}
```

```
public boolean full() {  
    return (topIndex >= capacity - 1) ;  
}  
}
```



# ג'אווה לא מחפשת דליים מתחת לכיור

- אני כן, ואם הייתי מנסה לעקוב אחרי הוראות ניקוי האקווריום, והייתי קורא שצריך דלי נקי, הייתי מחפש ומוצא

- אבל ג'אווה לא, ולכן, הקוד הבא יכשל ולא יעבור קומפילציה,

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

- ג'אווה רוצה שהלקוח יגיד איזה סוג עצם (מאיזו מחלקה) הוא רוצה לבנות; ג'אווה לא תנחש עבורו, אפילו רק מחלקה מממשת את טיפוס המנשק של המשתנה, או רק מחלקה אחת שממשת את המנשק ויש לה בנאי מתאים

- מה שצריך בג'אווה הוא

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

# אסימטריה

- שגרה כמו Find שהגדרנו לחיפוש גרסה מסוימת של מחרוזת יכולה להשתמש רק בטיפוס המנשק, ולכן לעבוד עם כל מימוש שיועבר לה כארגומנט
- אבל קוד שצריך ליצור עצמים לא יכול להשתמש בהם רק דרך טיפוס המנשק המתאימים, מכיוון שלאופרטור new חייבים להעביר שם של מחלקה, לא של מנשק

# דוגמא לשימוש במחסנית

```
public static void main(String[] args) {  
    Stack <String> s =  
        new LinkedStack <String> ();  
    if (!s.full())  
        s.push("hello");  
    if (!s.full())  
        s.push("world");  
    System.out.println(s.top());  
}
```

# ואם רוצים את המימוש האחר

- יש להחליף את

```
Stack <String> s =  
    new LinkedStack <String> ();
```

- ב

```
Stack <String> s =  
    new FixedCapacityStack <String> (10);
```

- יתר הקוד בלתי תלוי במימוש שנבחר
- במקרה זה הבנאים שונים בפרמטרים שלהם
- בכל מקרה, המנשק אינו קובע כיצד יראו הבנאים

# הפתרון: בתי חרושת (factories)

עצם שמממש מנשק יצירת עצמים מטיפוס מנשק מסוים

```
interface VersionedStringFactory {
    public VersionedString construct();
}
class LinkedVersionedStringFactory
    implements VersionedStringFactory {
    public VersionedString construct() {
        return new LinkedVersionString();
    }
}
```

# שימוש בבית חרושת (1)

```
class SomeClass {  
    private VersionedStringFactory f;  
    public SomeClass(VersionedStringFactory f,..)  
        { this.f = f; ... }  
    ...  
    public someMethod() {  
        VersionedString vs = f.construct();  
        vs.add("First version");  
        ...  
    }  
}
```

## שימוש בבית חרושת (2)

- בזמן יצירת העצם ששירותיו צריכים `VersionedString` חדשים, מעבירים לבנאי שלו בית חרושת; אפשר להעביר לו בית חרושת שייצר `LinkedVersionedString` או בית חרושת שייצר עצמים אחרים שמקיימים את המנשק
- בקוד של המחלקה הלקוחה אין אזכור לספקים ספציפיים, לכן היא יכולה לעבוד עם כל ספק, גם ספק שייכתב בעתיד
- ניתן כמובן גם להעביר את בית החרושת ישירות לשירות שהקוד שלו צריך עצמים חדשים
- המבנה הרצוי תלוי בעיקר בשאלה מי הקוד שמחליט באיזה ספק להשתמש, ולכן באיזה בית חרושת להשתמש; בדרך כלל, זהו קוד בקרת תצורה שרחוק למדי מהקוד המשתמש

# בתי חרושת משוכללים

בית חרושת יכול לייצר עצמים תוך שימוש בארגומנטים

```
interface VersionedStringFactory {
```

```
    public VersionedString construct();
```

*Ensures: returns a reference to a new VersionedString*

```
    public VersionedString construct(int m);
```

*Ensures: returns a reference to a new VersionedString  
that keeps at least the m most recent versions*

```
}
```



## אם החוזה מרשה, מותר להתבטל

```
class LinkedVersionedStringFactory
    implements VersionedStringFactory {
    public VersionedString construct() {
        return new LinkedVersionString();
    }
}
```

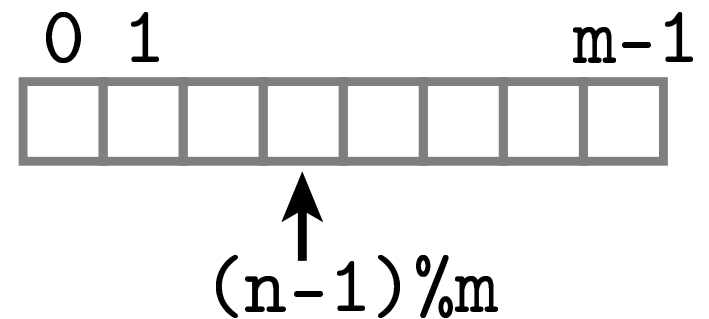
*keeps an unlimited number, so satisfies the contract*

```
public VersionedString construct(int m) {
    return new LinkedVersionString(); }
}
```

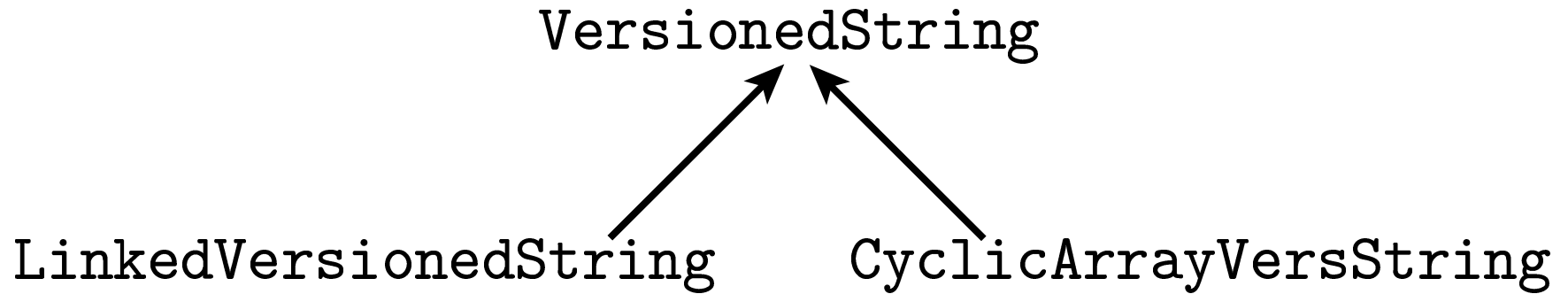
`construct(int)` מאפשר ללקוח לתת עצה לבית החרושת;  
בית החרושת לא חייב לנצל את העצה

# אבל אפשר להתאמץ

```
class SmartVersionedStringFactory
    implements VersionedStringFactory {
public VersionedString construct() {
    return new LinkedVersionString();
}
public VersionedString construct(int m) {
    return new CyclicArrayVersString(m);
}
```



# ניצני ארגון עבור טיפוסים



- בין שלושת הטיפוסים יש יחסים: שתי המחלקות מממשות את המנשק
- המנשק **יותר כללי** מהמחלקות שמממשות אותו
- הכלל הבסיסי: משתנה או שדה מטיפוס כללי יותר (כאן מנשק) יכול להתייחס לעצם מטיפוס יותר ספציפי
- אבל לא להיפך

## עוד דוגמה: מיון מערך

*requires: a != null && a's elements != null*

*ensures: a becomes sorted*

```
void insertionSort(Comparable[] a) {
    int i, j;
    for (j=1; j<a.length; j++) {
        Comparable key = a[j];
        for (i=j-1;
            i>=0 && a[i].compareTo(a[j])>0;
            i--) a[i+1] = a[i];
        a[i+1] = key;
    } }
```

# מנשק להשוואות

```
interface Comparable {  
    requires: other != null  
    ensures: return == 0 iff this = other  
        return == 1 iff this > other  
        return == -1 iff this < other  
    int compareTo(Comparable other);  
}
```

בספריות של השפה מוגדר מנשק `java.lang.Comparable` הוא דומה ברוחו ובמהותו למנשק שהגדרנו כאן, אבל לא זהה לו לגמרי; בהמשך נבין למה

# מימוש המונסק ב-VersionedString

```
class LinkedVersionedString
    implements VersionedString,
               Comparable {
    protected int      n;
    protected Version last;
    int compareTo(Comparable other) {
        if (this.n > other.n) return 1;
        if (this.n < other.n) return -1;
        return 0;
    }
}
```

# לפעמים אני מרצה ולפעמים הורה

- וזה נורמלי
- אני מספק שירותים אחרים בתור מרצה ואחרים בתור הורה
- אבל אני תמיד נשאר אני (וממש שני מנשקים שונים)
- יש מכונת פקס שלפעמים היא פקס, לפעמים מכונת צילום, לפעמים טלפון, ולפעמים משיבון
- הגדרה של מחלקה יכולה להצהיר שהיא מממשת מספר מנשקים

# אבל המימוש לא נכון!

```
int compareTo(Comparable other) {  
    if (this.n > other.n) return 1;  
    if (this.n < other.n) return -1;  
    return 0;  
}
```

- השירות למעשה מניח ש-`other` הוא מטיפוס `LinkedVersionString`
- אבל `other` הוא מטיפוס `Comparable`, לא מטיפוס `LinkedVersionString`, ולכן הקומפיילר לא ירשה להתייחס לשדה `n` דרכו



# ניסיון לפתרון הבעייה

נדרוש בתנאי הקדם של `insertionSort` שכל העצמים במערך יהיו מאותה מחלקה, ונתייחס ל-`other` בהתאם:

```
int compareTo(Comparable other) {  
    LinkedVersionOfString other_lvs  
    = (LinkedVersionOfString) other;  
    if (this.n > other_lvs.n) return 1;  
    if (this.n < other_lvs.n) return -1;  
    return 0; }  
}
```

האופרטור שבסוגריים נקרא יציקה (`cast`), והוא מייצר ייחוס מטיפוס נתון לעצם נתון, אם העצם מתאים לטיפוס

# המרת טיפוס ייחוס

- עצמים יכולים להיות מומרים בין טיפוס ייחוס שונים.
- המרה מרחיבה (widening) - מטיפוס ספציפי לטיפוס כללי יותר, תמיד מותרת באופן אוטומטי, למשל בהשמה

```
LinkedVersionedString lvs;
```

```
VersionedString vs;
```

```
lvs = vs; // vs is widened
```

- המרה מצרה (narrowing) - מטיפוס כללי לטיפוס ספציפי יותר, דורשת פעולה מפורשת של יציקה (cast), ולא תמיד מצליחה.

# יציקות (casts)

• תחביר:

( <TypeToBeConvertedTo> ) <Expression>

• הביטוי <Expression> מחושב, ונעשה ניסיון להמיר את ערכו לטיפוס <TypeToBeConvertedTo>

• כאשר הביטוי הוא ייחוס, היציקה מצליחה אם הייחוס מתייחס לעצם מתאים לטיפוס שיוצקים לתוכו

• יציקה למטה (downcast): יציקה של ייחוס לטיפוס פחות כללי; כרגע הכוונה ליציקה של ייחוס למנשק לייחוס למחלקה שמממשת את המנשק; בהמשך נראה שיש עוד מקרים

# יציקות (casts) - המשך

- יציקה למעלה (upcast): יציקה של ייחוס לטיפוס יותר כללי, למשל יציקה של ייחוס למחלקה לייחוס למנשק שהמחלקה מממשת; שוב, בהמשך נראה עוד מקרים
  - יציקה למעלה תמיד מצליחה, (כאמור לא נדרש אופרטור יציקה מפורש); היא פשוט גורמת לקומפיילר לאבד מידע
  - יציקה למטה עלולה להיכשל; בהמשך נראה מה קורה אז
  - אפשר לצקת גם ערכים פרימיטיביים. לדוגמא, ערך הביטוי הוא מספר בנקודה צפה והוא מומר לשלם
- $$(int) (x + 2.5 * y)$$

# אבל היציקה לא פתרה את הבעיה

- כי הלקוח לא בהכרח יודע אם כל העצמים שהוא רוצה למיין שייכים לאותה מחלקה או לא
- למשל, אם `SmartVersionedStringFactory` ייצר את העצמים, יתכן שהלקוח קיבל עצמים מכמה מחלקות
- אם בעיני הלקוח אפשר למיין עצמים שמממשים `VersionedString`, אז סימן שהמיון צריך לבטא תכונות של העצמים שהלקוח מודע להם, לא את המימוש הנסתר
- אי לכך, צריך להצהיר שכל עצם שמממש `VersionedString` מממש גם `Comparable`

# מנשק מבטיח לממש מנשק אחר

```
interface VersionedString
    extends Comparable {
    public void    add(String s)        ;
    public int    length()              ;
    public String getLastVersion()     ;
    public String getVersion(int i)    ;
}
```

```
class LinkedVersionedString
    implements VersionedString { ... }
```

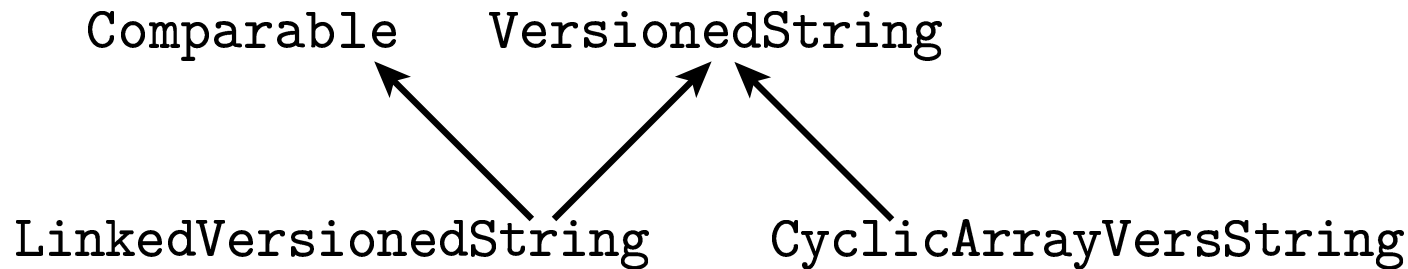
- המנשק לא צריך להצהיר על `compareTo`, קיום השירות מובטח מעצם העובדה שהמנשק מרחיב את `Comparable`

## והמימוש הנכון...

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

קצת פגום: למרות שהמימוש לא ספציפי למחלקה צריך לשכפלו בכל מימוש של `VersionedString`; בהמשך נתקן

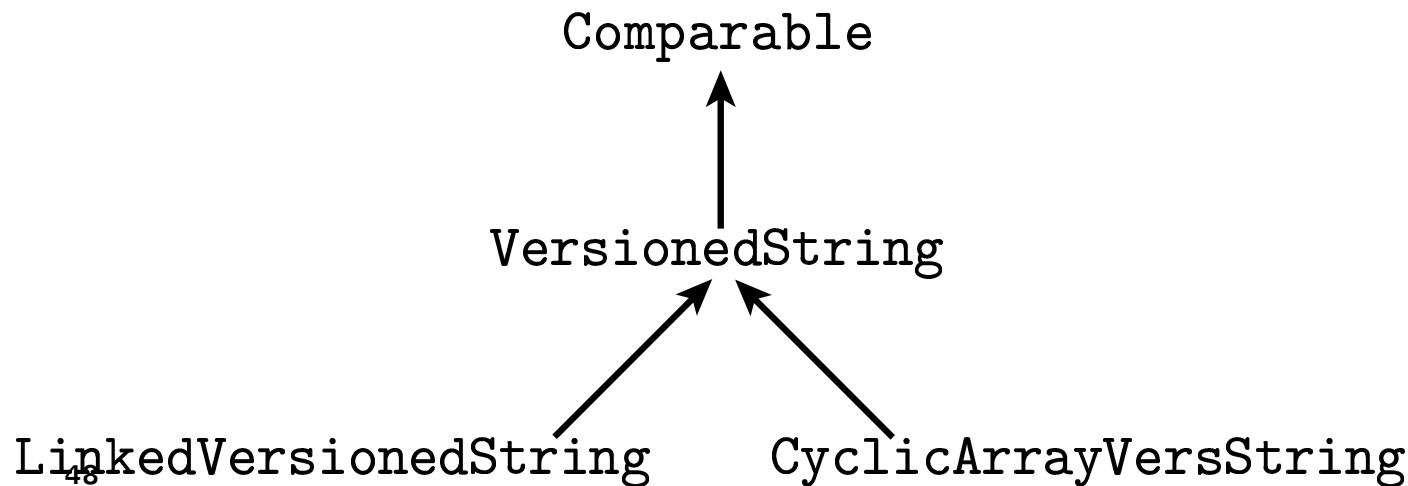
# היררכית הטיפוסים גדלה



זה לא היה  
מבנה מוצלח  
במקרה זה,  
אבל הוא  
הראה

שהיחסים בין טיפוסים הם גרף א-ציקלי מכוון

המבנה הזה היה יותר מוצלח, והוא מראה שגרף היחסים עשוי להיות עמוק





# סיכום מנשקים

- שימוש בטיפוס מנשק מאפשר ללקוח להצהיר שייחוס (משתנה או שדה) מתייחס לעצם שמספק שירותים מסוימים, בלי לציין מאיזו מחלקה העצם; זה מאפשר כלליות בלקוח
- לקוח כזה נקרא רב-צורתני (polymorphic)
- שימוש בבית חרושת (factory) מאפשר ללקוח לבנות עצמים עם מנשק מסוים בלי לציין בעצמו מאיזו מחלקה הם
- מחלקה יכולה לממש מספר מנשקים
- מנשק יכול להרחיב מנשק אחר או מספר מנשקים אחרים
- רב צורתיות מאפשרת ניצול של קוד קיים במקרים נוספים ומונעת שכפול של קוד, שכפול שהוא יקר לפיתוח ותחזוקה