

חלק 7

מחלקות ושירותים
מוכללים (generics)

ננסה להכליל את `VersionedString`

- השירותים שהמחלקה הגדירה (או המנשק 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 getLastError() {...}  
}
```

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

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));
```

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)

שימוש במחלקה מוכללת

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

- השירות `vs.add` כאשר `T`
הוא `String`, ובאופן דומה לגבי `vs.getVersion`

לפני הפרטים, העיקר

- מנגנון ההכללה מיועד לאפשר שימוש חוזר בקוד בלי לאבד מידע לגבי הטיפוס הסטאטי 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 הסטאטי 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

איך זה עובד

- הקומפילר ממפה את כל המחלקות המוכללות `<Something>Versioned` למחלקה אחת רגילה (לא מוכללת) `<Object>Versioned`
- בקוד שמתמש במחלקה מוכללת, הקומפילר מוסיף לקוד יציקות על מנת לבצע השמות מ-`Object` לטיפוס הספיציפי, למשל `String`

- הקומפילר מוודא שהיציקה תמיד תצליח ולעולם לא תודיע על `ClassCastException`,

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

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

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

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>`

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<?>`

כנסה 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 שלה תהיה הסיפא של b

• **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

הפותרון: שירות מוכלל

```
<T> static 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 המשווא-אים להיות אותו משהו, ולכן הערך המוחזר מ-`getVersion` ב `b` הוא מטיפוס שמתאים לארגומנט של `a.add`
- שירות מוכלל, כמו שירות של מחלקה מוכללת, יכול גם להחזיר ערך מטיפוס `T` ולהגדיר משתנים מקומיים מטיפוס `T`

הגבלת הכלליות

- כעת ננסה להגדיר פרוצדורה שבודקת האם הגרסאות ששמורות בעצם הן בסדר מונוטוני עולה, לא מבחינת האינדקס שלהן אלא מבחינת תוכן

• למשל, הסדרות (1,3,4,898) ו-(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; }  
}
```

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)

לא הסתבכנו מדי?

• ברור שכן

• הרי גם בעזרת הכלים שהיו לנו לפני שהצגנו את מנגנון ההכללה יכולנו להגדיר את `monotonic`

• היינו מגדירים מנשק `Versionable` על מנת לייצג את האיברים של סדרת גרסאות, והיינו מגדירים אותו כמנשק שמרחיב את `Comparable`, ואולי עוד מנשקים

• 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

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 {}
```

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)?

הגבלת כלליות עם הסם 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 שהמשווה צריך להיות מסוגל להשוות עצמים מאיזשהו טיפוס `T` ש-`Double` הוא סוג שלו,
- ```
int sort(E[] a, Comparator<? super E> {...})
```

# 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 = same as Versioned<?>
```

```
new Versioned(); same as Versioned<Object>
```

```
raw = vs; ok
```

```
vs = raw; "unchecked" compiler warning
```

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

# מהירות

- בגלל שבג'אווה הכללה ממומשת באמצעות מנגנון המחיקה, בזמן ריצה אין זכר לפרמטר הטיפוס

- כלומר, בזמן ריצה אי אפשר להבחין בין עצם מטיפוס `<String>Versioned` ובין עצם מטיפוס

`<Integer>Versioned`, 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
```

# סיכום generics

- מנגנון ההכללה מאפשר להימנע מיציקות בלי לשכפל קוד
- קוד שאין בו יציקות מפורשות ושאין בו טיפוסים נאים (ליתר דיוק, אם הקומופיילר לא הזהיר לגבי השימוש בטיפוסים נאים) הוא בטוח מבחינת טיפוסים (type safe)
- קוד כזה לא יכשל בביצוע יציקה בזמן ריצה: הבדיקות מועברות לזמן הקומופיילציה
- 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 מוזריות