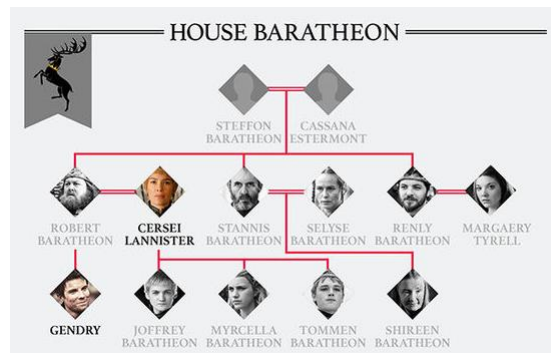


תוכנה 1 בשפת Java

שיעור מספר 12: "ירושה מתקדמת"

(הורשה III)



היום בשיעור

- עוד על Generics
- תבניות והורשה
- קבלנות משנה (הורשה והחזקה)
- שימוש לרעה בהורשה



עוד על Generics

מה עושים ללא מחלקות גנריות

- אחת הדוגמאות השכיחות לשימוש בהמרת טיפוסים ב Java היא השימוש במבני נתונים לפני Java 1.5
- מכיוון שעד לגרסה 1.5 לא ניתן היה להשתמש בטיפוסים מוכללים (generics), נאלצו כותבי הספריות להניח שהאברים הם מהמחלקה הכללית ביותר, כלומר Object
- נניח כי רוצים לכתוב מנשק ו/או מחלקה עבור מחסנית, שתאפשר ליצור מחסנית של שלמים, מחסנית של מחרוזות, וכו' **ללא שימוש ב Generics**
- בדוגמא – מנשק למחסנית, ומחלקה מממשת (ללא החוזה)

ממשק מחסנית

```
interface Stack {  
    public Object top ();  
    public void push(Object t);  
    public void pop();  
    public boolean empty();  
    public boolean full();  
}
```

מימוש מחסנית פשוט

```
public class FixedCapacityStack implements Stack{
```

```
    private Object [] content;  
    private int capacity;  
    private int topIndex;
```

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

```
    public Object top () {  
        return content[topIndex];  
    }
```

מימוש מחסנית פשוט

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

public void pop() {
    topIndex--;
}

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

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

איך נשתמש במחסנית?

■ נניח שרוצים מחסנית של מחרוזות:

```
Stack s = new FixedCapacityStack(5);  
s.push("hello");  
String t1 = s.top();           // compilation error  
String t2 = (String) s.top();  // ok
```

■ באחריות המתכנתת לוודא שכל האברים המוכנסים למחסנית הם מאותו טיפוס (כאן מחרוזות), אחרת ה Casting ייכשל.

```
Stack s = new FixedCapacityStack(5);  
s.push("hello");  
s.push(new Integer(4));  
s.push(new PolarPoint(3,2));  
String t2 = (String) s.top();  // compilation ok. Runtime Error !
```


בטיחות טיפוסים

- מכיוון שבדיקת ההמרה נעשית בזמן ריצה אנחנו מאבדים בטיחות טיפוסים

- זהו דבר שאינו רצוי – אנו מעוניינים להעביר בדיקות רבות ככל הניתן לזמן קומפילציה
 - מדוע?

- פתרון אחר: מנשק/מחלקה נפרדת לכל טיפוס איבר – שכפול קוד!

- הוספת הטיפוסים המוכללים לשפה פותרת גם את בעיית בטיחות הטיפוסים וגם את בעיית שכפול הקוד

מחלקה מוכללת (גנרית)

- 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 המרה
- תפקיד ההכללה הוא למנוע צורך בהמרות, שנבדקות מאוחר
- אבל העניינים מסתבכים בגלל האינטראקציה בין מנגנון ההכללה ובין יחס ההורשה (יחס ה-is-a)
- קושי נוסף: תאימות בין גרסאות גנריות ולא גנריות

איך זה עובד?

- הקומפיילר ממפה את כל המחלקות המוכללות `FCStack<Something>` למחלקה אחת רגילה (לא מוכללת) `FCStack<Object>` שהיא בעצם

- בקוד שמשמש במחלקה מוכללת, הקומפיילר מוסיף לקוד המרות על מנת לבצע השמות מ-`Object` לטיפוס הספיציפי, למשל `String`

- הקומפיילר מוודא שההמרה תמיד תצליח ולעולם לא תודיע על `:ClassCastException`

```
String t = (String) s.top();
```

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

בטיחות טיפוסים

```
Stack <String> ss = new FCStack <String> (5);  
✓ ss.push("The letter A");  
✗ ss.push(new Integer(3));  
✓ String t = ss.top(); // same as: (String)ss.top();
```

מכיוון שרק מחרוזות יכולות להיות מוכלות במחסנית אין עוד צורך בהמרה ■

```
Stack <Rectangle> sr = new FCStack <Rectangle>(5);  
Rectangle rr = new Rectangle(...)  
Rectangle rc = new ColoredRectangle(...)  
ColoredRectangle cc = new ColoredRectangle(...)  
  
✓ sr.push(rr);  
✓ sr.push(rc);  
✓ sr.push(cc);
```

הכללה ויחס is-a

```
Stack <String> ts = new FCStack <String> (5) ;
```

```
Stack <Object> to = new FCStack <Object> (5) ;
```



```
to = ts ;
```



```
ts.push("The letter A") ;
```



```
ts.push(new Integer(3)) ;
```



```
to.push(new Integer(3)) ;
```

מסקנה: `FCStack<String>` אינו סוג של `FCStack<Object>` ■

זה לא אינטואיטיבי אבל נכון. ■

יחס is-a במערכים

■ האם מתקיים יחס is-a בין מערך של מחרוזות למערך של אובייקטים?

```
String [] strArr = new String[5];  
Object [] objArr = strArr;
```

■ השמה זו חוקית מבחינה תחבירית בלבד. `objArr` מצביע למערך של מחרוזות, ולכן שימוש שגוי בו יגרום לשגיאת זמן ריצה.

```
objArr[0] = new Integer(); // throws ArrayStoreException
```

■ ההשמה הבאה גם היא מתקמפלת, אך גורמת לשגיאת זמן ריצה:

```
Object[] objArr = new Object[1];  
String[] strArr = (String[]) objArr; // throws ClassCastException
```

השימוש בטיפוסים מוכללים סותם פרצה זו בתחביר המקורי של שפת Java ומונעת מקרים כאלה כבר בשלב הקומפילציה.

מערכים מוכללים

■ Java מאפשרת לנו להגיד מערך עם טיפוס גנרי, אבל לא מאפשר לייצר מערך כזה, בגלל מחיקת הטיפוסים בזמן ריצה:

```
public class Test<T>{  
    ✓ T[] arr;  
  
    public Test(){  
        ✗ arr = new T[10];  
    }  
}
```

■ טיפוסים מוכללים "לא מסתדרים" עם מערכים ב Java. בד"כ מומלץ להימנע משימוש במערכים מוכללים, ולעבוד עם אוספים מוכללים במקום זאת.

מערכים מוכללים

■ אפשר להתחכם:

```
arr = (T[])new Object[10];
```

■ "טריק" כזה יעבוד עבור קוד המבוצע בתוך המחלקה המוכללת

```
public T[] createArr(T t){  
    arr = (T[])new Object[10];  
    arr[0] = t;  
    return arr;  
}
```

■ אבל כשנרצה להחזיר את המערך, נקבל שגיאת זמן ריצה, כיוון שהמערך המוחזר הוא למעשה מערך של Object

```
Test<String> test = new Test<>();  
String[] arr = test.createArr("abc"); // throws ClassCastException
```


טיפוסים נאים (raw types)

- מנגנון ההכללה נוסף לג'אווה מאוחר, ולכן היה צורך לאפשר שימוש במחלקות פרמטריות גם מקוד ישן שאין בו הכללות

```
class FCStack <T> implements Stack <T> {...}
```

...

```
Stack <String> vs = new FCStack <String> ();
```

```
Stack raw = new FCStack ();
```

```
raw = vs; // ok
```

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

- בשימוש בטיפוס נא, פרמטר הטיפוס מוחלף ב"גבול העליון" (בדרך כלל Object)

הגבול הוא השמיים

- גבול עליון הוא שם של המחלקה או המנשק שממנה יורש הטיפוס הפרמטרי
- כאשר הגבול העליון הוא Object לא ניתן לבצע כל פעולה על עצמים מהטיפוס הגנרי
- על כן, בהגדרת טיפוס גנרי ניתן לספק גבול עליון אחר
- הדבר מאפשר להשתמש בגוף המחלקה הגנרית בשירותים המוגדרים באותו גבול עליון ללא צורך בהמרה

```
public class SortedSetImplementation<T extends Comparable> {  
    ...  
    T elem1 = ...  
    T elem2 = ...  
    ... elem1.compareTo( elem2) ....  
    expectComparable(elem1);  
}
```

Comparable גנרי

■ ראינו דוגמאות של המנשק Comparable בגירסה נאה (raw)

■ השימוש בה בעייתי

■ יתכנו שני עצמים שכל אחד מהם Comparable אבל הם אינם Comparable זה לזה

■ לדוגמא: Integer ו-String

■ אנחנו נעדיף את הגירסה הגנרית, שהשימוש בה הוא:

```
public class MyClass implements Comparable<MyClass> {  
    public int compareTo(MyClass other) {  
        ...  
    }  
}
```

■ בצורה זאת מגדירים מחלקה שעצמיה ברי השוואה לעצמם, ומספקים שרות שמבצע את ההשוואה

■ אם רוצים אפשרות השוואה למחלקה כללית יותר, זה נעשה יותר מסובך (לא נעסוק בזה בקורס)

מוזרויות

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

- כלומר, בזמן ריצה אי אפשר להבחין בין עצם מטיפוס `FCStack<Integer>`, `FCStack<String>` ובפרט, בזמן ריצה נראה ששניהם מאותה מחלקה

- זה משפיע על בדיקת שייכות למחלקה (`instanceof`), על המרות של עצמים מוכללים, ועל שדות המסומנים `static`

- וזה מונע אפשרות לקרוא לבנאי על פי פרמטר טיפוס, כלומר:

```
<T> void m(T x) { T y = new T(); ... } // illegal
```

- **ויש עוד הרבה מזה...**

למשל...

- רצינו לשלב את הקוד הבא (שמצאנו בגרסה ישנה של המוצר) במוצר החדש:

```
public static void printList(List list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i % 2 == 0) {
            System.out.println(list.get(i));
        }
    }
}
```

- כדי להימנע מאזהרות קומפילציה נשנה את List לטיפוס מוכלל:

```
public static void printList(List<Object> list) {
    for(int i=0, n=list.size(); i < n; i++) {
        if (i % 2 == 0) {
            System.out.println(list.get(i));
        }
    }
}
```

- לא טוב, לא ניתן להעביר לשרות List<String>



ג'וקרים

■ נשתמש בג'וקר (סימן שאלה - ?)

```
public static void printList(List<?> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i % 2 == 0) {  
            Object obj = list.get(i);  
            System.out.println(obj);  
        }  
    }  
}
```



ג'וקרים

■ כדי שנוכל לבצע פעולות על אברי הרשימה יש לספק חסם עליון, כמו בשרות:

```
public static double sumPerimeters(List<? extends IShape> list) {  
    double total = 0.0;  
    for(IShape n : list)  
        total += n.perimeter();  
    return total;  
}
```

■ משמעות ההגדרה: הטיפוס הגנרי של `list` הוא טיפוס המרחיב את `IShape`, כולל `IShape` עצמו כמובן.

■ שימו לב לשימוש ב `extends` גם עבור מנשקים. זהו תחביר מיוחד להרחבות.

■ שימוש בשירות: `List<IShape> shapes = ...`

`List<Circle> circles = ...`

`List<Triangle> triangles = ...`

`double shapesPerimeterSum = sumPerimeters(shapes);`

`double circlesPerimeterSum = sumPerimeters(circles);`

`double trianglesPerimeterSum = sumPerimeters(triangles);`



ג'וקרים

יש גם חסמים תחתונים:

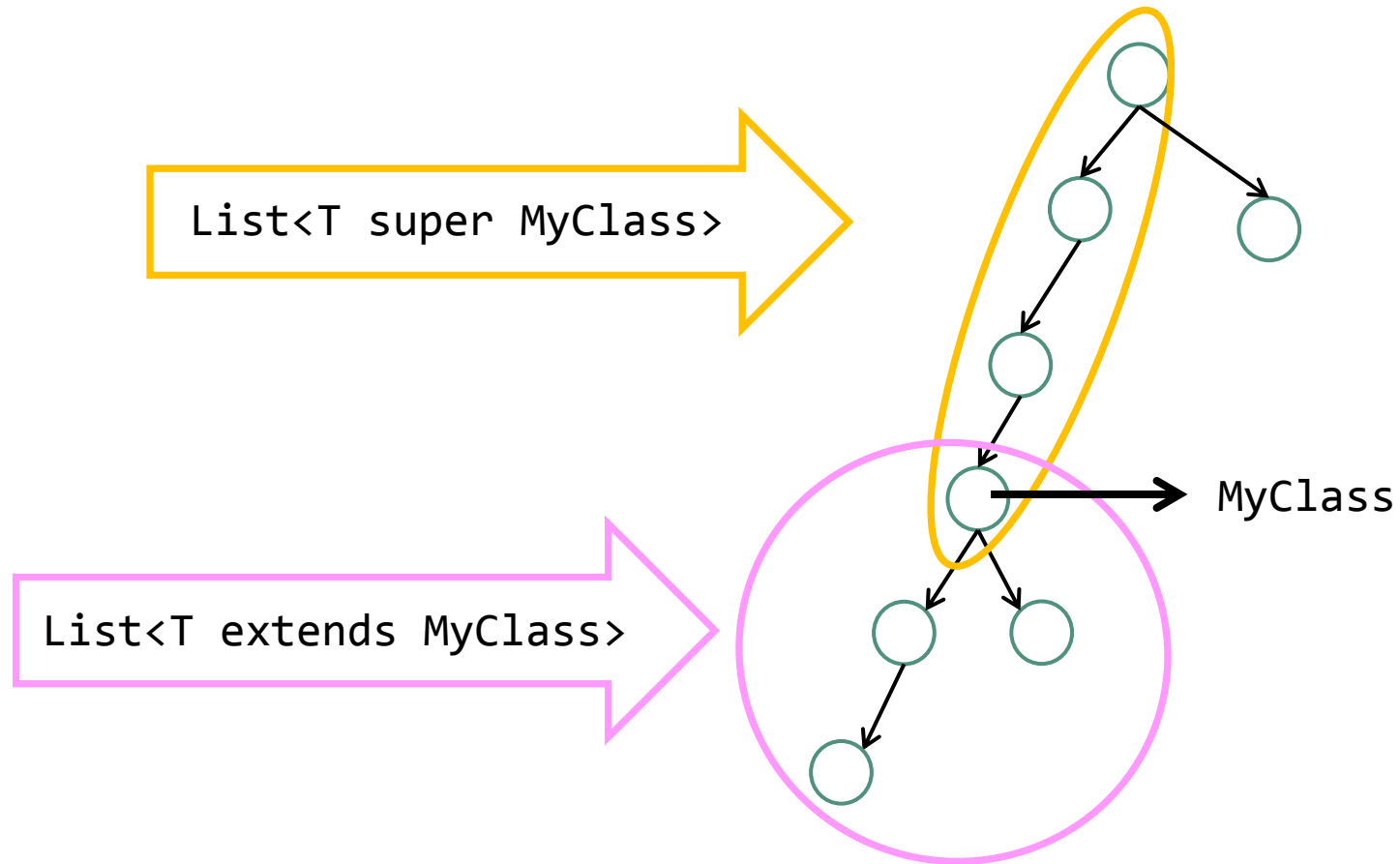
```
public static boolean addItem(List<? super ColoredRectangle> lst,
                               ColoredRectangle item) {
    return lst.add(item);
}
```

המשמעות: הטיפוס הגנרי של הרשימה list הוא ColoredRectangle או טיפוס שאותו ColoredRectangle מרחיב.

שימוש בשירות:

```
List<ColoredRectangle> cRectangles=...;
List<Rectangle> rectangles=...;
List<Object> objects=...;
ColoredRectangle cRect=...;
addItem(cRectangles, cRect);
addItem(rectangles, cRect);
addItem(objects, cRect);
```


super vs. extends



שירותים מוכללים

- ניתן להגדיר טיפוס גנרי עבור שירות בודד, ולא רק למחלקה.

```
public class Test{
    public static void main(String[] args){
        List<Integer> intLst = Arrays.asList(1,2,3);
        List<String> strLst = Arrays.asList("a", "b", "c");
        int firstInt = getFirstItem(intLst);
        String firstStr = getFirstItem(strLst);
    }

    /*
     * @pre list.size() > 0
     */
    public static <T> T getFirstItem(List<T> list){
        return list.get(0);
    }
}
```

השירות `getFirstItem` מכיל פרמטר גנרי `<T>`. ערכו של הפרמטר הגנרי יקבע בזמן הקריאה לשירות. למשל, כאשר נשלח `List<String>` כפרמטר, ערכו של `T` יקבע ל `.String`.

ובהקשר של מחלקות פנימיות...

```
public class MyType<E>{  
  
    class Inner{}  
    static class Nested{}  
  
    public static void main(String[] args) {  
        MyType mt; //warning: MyType is a raw type  
        MyType.Inner inn; //warning: MyType.Inner is a raw type  
        MyType.Nested nest; //no warning, not a parametrized type  
        MyType<Object> mt1; //no warning  
        MyType<?> mt2; //no warning, ? is OK for a type  
    }  
}
```

למה טוב שהקומפיילר שומר?

```
List names = new ArrayList(); //warning: raw type
names.add("Kyle");
names.add("Eric");
names.add(Boolean.FALSE);
```

```
for (Object o : names){
    String s = (String)o;
    System.out.println(s.toUpperCase());
}
//throws ClassCastException
// java.lang.Boolean cannot be cast to java.lang.String
```

גילוי השגיאה
בזמן קומפילציה
ולא בזמן ריצה!

```
List<String> names = new ArrayList<>();
names.add("Kyle");
names.add("Eric");
names.add(Boolean.FALSE); //compilation error!
```

<Object> מ Raw שונה

```
public static void main(String[] args){
    List<String> strLst = new ArrayList<>();
    appendNewObject(strLst); //compilation error!
}
```

```
public static void appendNewObject(List<Object> lst){
    lst.add(new Object());
}
```

מה היה קורה אם הפונקציה `appendNewObject` היתה מקבלת `List` נא?

Raw שונה מ <?> (wildcard)

```
public static void main(String[] args){
    List<String> strLst = new ArrayList<>();
    appendNewObject(strLst); //this is fine
}
public static void appendNewObject(List<?> lst){
    lst.add(new Object()); //compilation error!
}
```

כמובן שזה לא הגיוני שיהיה ניתן להוסיף עצם מטיפוס Object לרשימה של מחרוזות, לכן, צריך למנוע את זה כבר בשלב הקומפילציה.

<?> כמחלקת בסיס

```
public static void printCollection(Collection<?> c){
    for (Object o: c){
        System.out.println(o);
    }
}
```

- ניתן לשלוח לפונקציה printCollection כל אוסף.
- בתוך printCollection ניתן לגשת לאלמנטים מתוך c ולשייך להם את הטיפוס הסטטי Object.

```
Collection<?> c = new ArrayList<>();
c.add(new Object()); // Compile time error
```

סיכום generics

- מנגנון ההכללה מאפשר להימנע מהמרות בלי לשכפל קוד
- קוד שאין בו המרות מפורשות ושאין בו טיפוסים נאים (ליתר דיוק, אם הקומפיילר לא הזהיר לגבי השימוש בטיפוסים נאים) הוא בטוח מבחינת טיפוסים (type safe)
- קוד כזה לא יכשל בביצוע המרה בזמן ריצה: הבדיקות מועברות לזמן הקומפילציה
- 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 מוזרויות (ועוד לא דיברנו על כולן...)
- דיון מקיף (מעניין, וברור) בנושא ניתן למצוא בפרק 4.1 של:
Java in a Nutshell, 5th Edition By David Flanagan

תבניות ויורשה

אלגוריתם כללי

Template Method Design Pattern

- מחלקות מופשטות (abstract) מגדירות שני סוגים של מתודות
 - מתודות ממשיות (effective, concrete)
 - מתודות מופשטות (abstract, deferred)
- ניתן להבחין בין רמות ההפשטה של שני הסוגים
 - המתודות הממשיות מגדירות רעיון כללי, תבניתי
 - המתודות המופשטות מגדירות אבני בניין (hooks) שבעזרתן ניתן לממש אלגוריתמים כלליים. המימוש של המתודות המופשטות נעשה במחלקות יורשות.
 - שימו לב – הטרמינולוגיה הפוכה!
- דוגמא: מימוש המתודה changeTop במחסנית לא מחייב הכרות עם מימוש המחסנית עצמה

מחסנית מופשטת

```
abstract class AbstStack <T> implements IStack<T> {  
  
    public void changeTop(T t) {  
        pop ();  
        push(t);  
    }  
  
    abstract public void push(T t);  
    abstract public void pop();  
}
```

- השרות changeTop אינו תלוי במימוש של push או pop אלא רק בחוזה שלהם
- changeTop מכונה אלגוריתם כללי
- pop ו-push הם hooks שמחלקות יורשות צריכות לממש בצורה ספציפית

ירושה ממחסנית מופשטת

מחלקות היורשות מ `AbstStack` צריכות רק לממש את ה `hooks` (שהוגדרו `abstract`), ומקבלות "בחינם" את האלגוריתמים הכלליים

```
class StackImpl<T> extends AbstStack <T> {  
    public void push(T t) {...}  
    public void pop() {...}  
}
```

דוגמאות נוספות:

- שימוש באיטרטורים למציאת מאפיינים של מבנה נתונים
- השרותים `distance` ו-`toString` של `AbstPoint`
- זה מאפשר בין היתר לתוכנת מערכת לקרוא לקוד של המשתמש (מחלקה שהמשתמש כתב, שיורשת ממחלקה של המערכת)

זוהי תבנית עיצוב **design pattern** – השימוש בה מדגיש שימוש מסוים של ירושה:

- היורש אינו מוסיף פעולות לטיפוס הנתונים (כמו למשל מלבן צבעוני שהוסיף את תכונת הצבעוניות למלבן), אלא מממש (`concretization`) אותו בדרך מסוימת
- למרות שהמימוש אינו ידוע במחלקת הבסיס (האבסטרקטית), כן ניתן לממש בה את האלגוריתם הכללי

הורשה מרובה

- מנגנון ההורשה נועד לתאר בצורה נכונה יחסים בין מחלקות המבטאות ישויות (טיפוסים) בעולם האמיתי
- לפעמים יש הצדקה להורשה מרובה. לדוגמא:
 - **עוזר הוראה** הוא גם **סטודנט** (תלמיד מחקר) וגם **איש סגל** (חבר בארגון הסגל הזוטר)
 - היחס is-a מתקיים עבור 2 ה'כובעים' של עוזר ההוראה ולכן הוא אמור לרשת ממחלקות שמייצגות את שני התפקידים
 - זו אינה בעיה תיאורטית - למתרגל שני כרטיסי קורא בספריה (סטודנט וסגל) ובכל אחד מהם מוענקות לו זכויות השאלה שונות

הורשה מרובה – עוד דוגמא

■ מספר ממשי (REAL) הוא גם מספרי (NUMERIC) וגם בן השוואה (COMPARABLE)

```
class NUMERIC {  
    ...  
    NUMERIC add (NUMERIC other);  
    NUMERIC subtract (NUMERIC other);  
}
```

```
class COMPARABLE {  
    ...  
    boolean lessThan (COMPARABLE other);  
    boolean lessThanEqual (COMPARABLE other);  
}
```

■ ולכן הגיוני אולי שיירש משתיהן:

```
class REAL extends NUMERIC , COMPARABLE {  
    ...  
}
```

שגיאת קומפילציה
ב Java אין דבר כזה!

אין ב Java הורשה מרובה

- אין ב Java הורשה מרובה (ואולי טוב שכך?)
 - אמא יש רק אחת
 - יש לעשות פשרות כואבות
- קיימות כמה תבניות עיצוב אשר מתמודדות עם הבעיה הזו בהקשרים שונים
- נתבונן באחת התבניות שממנה נוכל להשליך על אחת הדרכים לפתרון בעיית ההורשה המרובה
- **Bridge Design Pattern** – פיתוח מערכת מחלקות היררכית, כאשר לאחת המחלקות צאצאים מסוגים שונים

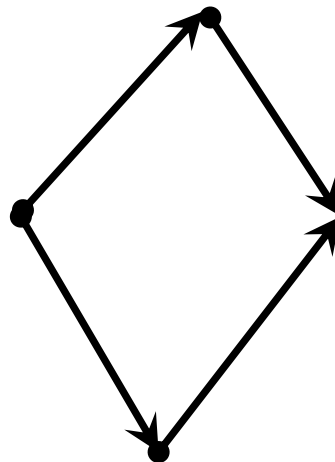
מה הבעיה בירושה מרובה?

```
class GoodDriver implements Driver {  
    boolean signalBeforeTurns()  
    {return true;}  
}
```

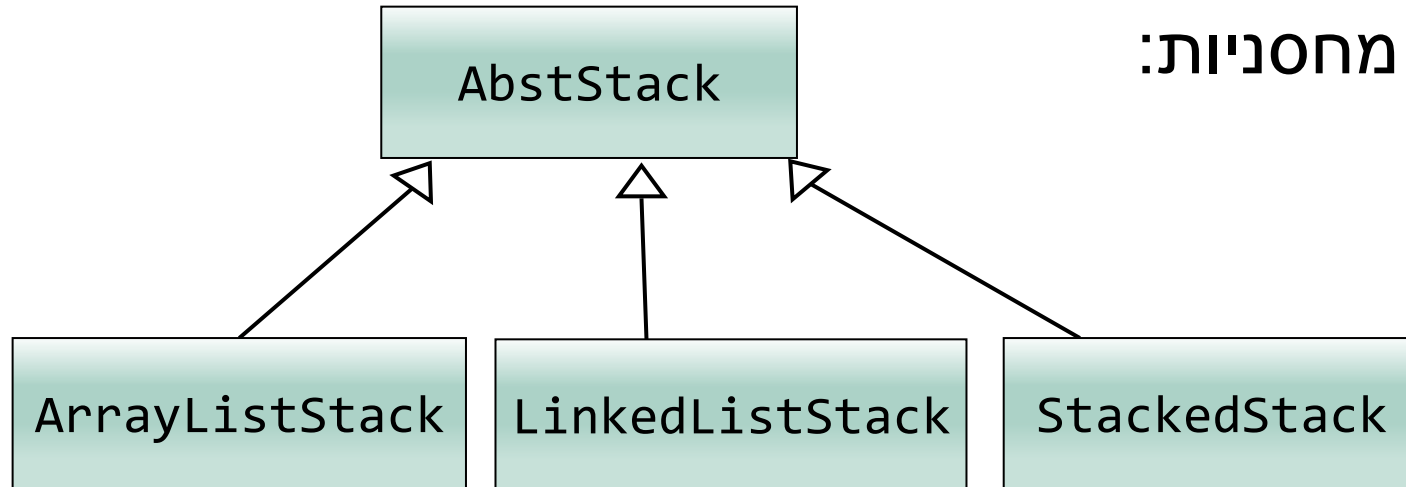
```
interface Driver {  
    ...  
}
```

```
Class OpportunisticDriver  
extends GoodDriver, BadDriver  
(not possible)
```

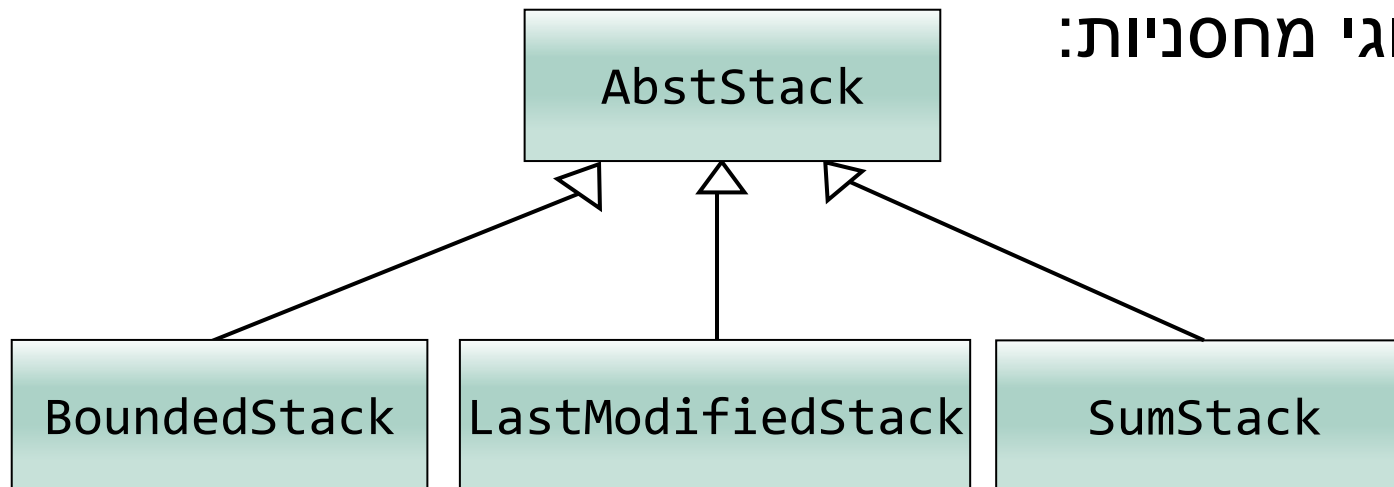
```
class BadDriver implements Driver {  
    boolean signalBeforeTurns()  
    {return false;}  
}
```



סוגי מחסניות:

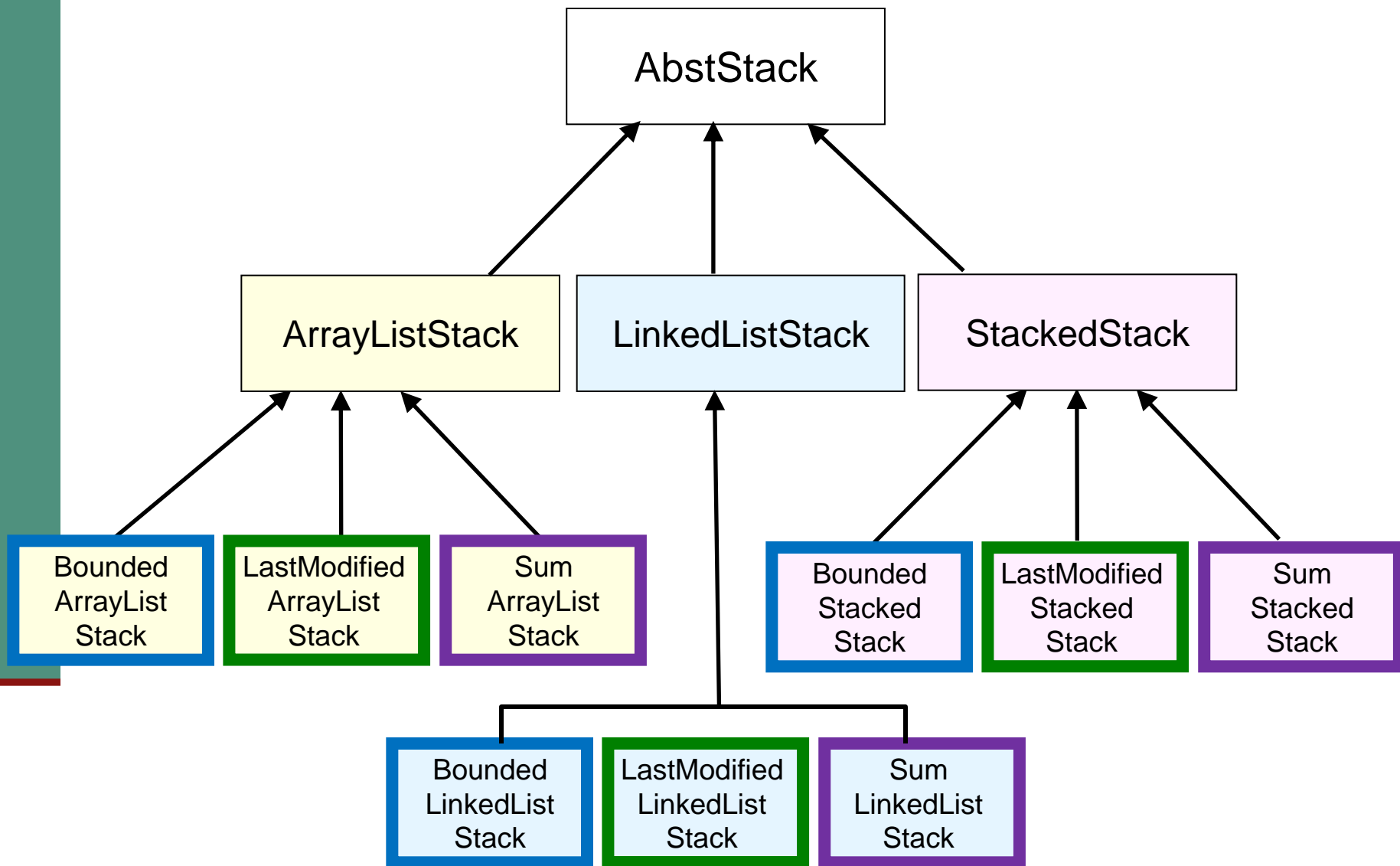


עוד סוגי מחסניות:



ילדים זה שמחה?

- סוג ההורשה של 3 המחלקות העליונות שונה מסוג ההורשה של 3 המחלקות התחתונות
- מה יקרה אם נרצה למשל: `SumArrayListStack` ?
- בשפות מסוימות (כגון C++ או Eiffel) ניתן ליצור מחלקה חדשה היורשת משתיהן
 - הדבר פותח פתח למכפלה קרטזית (9 מחלקות!) שתבטא את כל הצירופים האפשריים
 - דבר זה ייצור אינפלציה של מחלקות
- איך נממש זאת ע"י הורשה (לדוגמא את `SumArrayListStack` ב Java ?

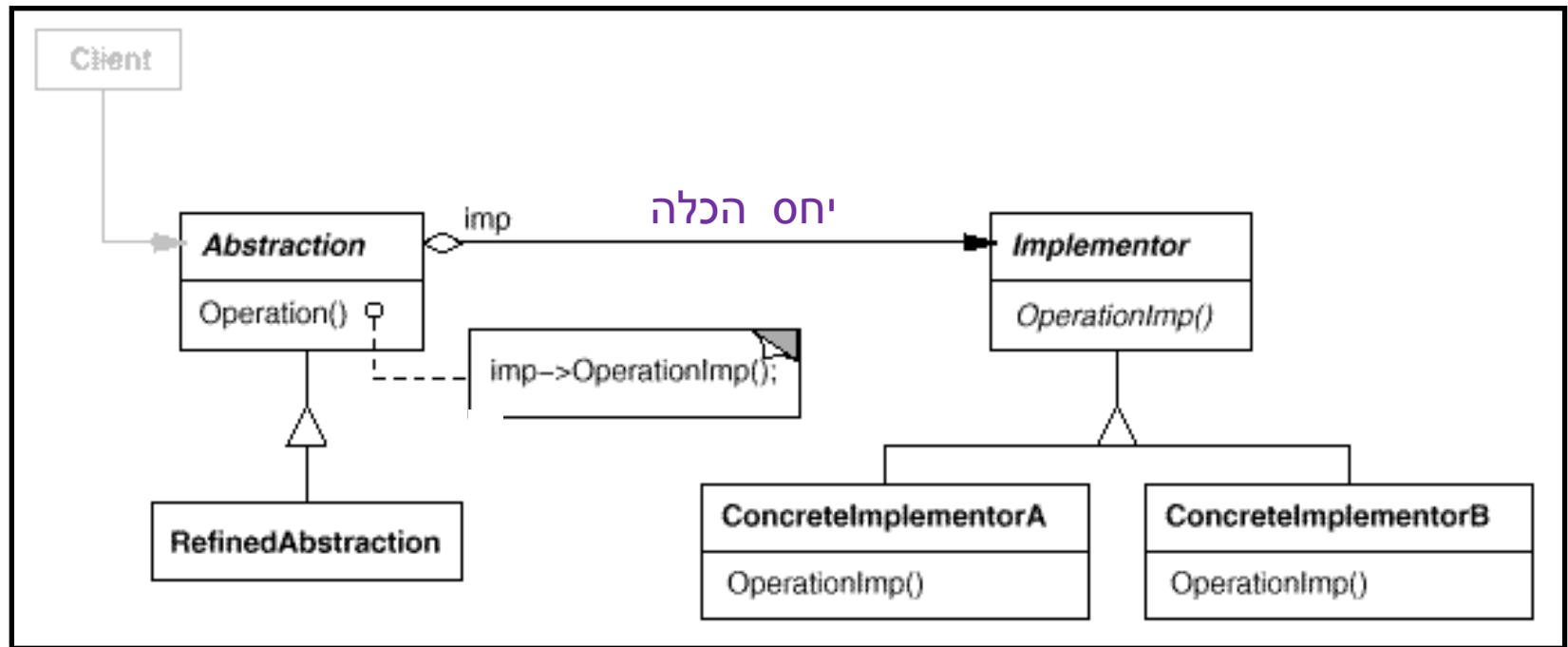


לא כל כך שמחה

- חסרונות:
 - שכפול קוד נורא
 - מה יקרה אם נרצה להוסיף טיפוס חדש כגון `TwoWayStack`?
 - צריך יהיה להוסיף אותו לכל תתי העצים
- גם הוספת הורשה מרובה לשפה לא הייתה פותרת את ההיררכיה הבעייתית
- הפתרון המוצע ע"י **תבנית העיצוב Bridge** היא המרת ירושת המימוש **בהכלה** (עם האצלה **delegation**)
 - פתרון זה מופיע בתבניות עיצוב רבות אחרות
- עצי ההורשה בשני המישורים (המופשט והמימושי) לא מתמזגים (אורתוגונליים)

Bridge Design Pattern

- תרשים מחלקות -



עיצוב באמצעות תבנית Bridge

מתאר התנהגות (LastModified ,Bounded)

```
public interface IStack<T> {  
    public void push (T e);  
    public void pop ();  
    public T top ();  
}
```

מתאר מימוש (ArrayList, LinkedList)

```
public interface IStackImpl<T> {  
    public void insert(T e);  
    public void remove();  
    public T get(int index);  
}
```

```
public interface IStack<T> {  
    public void push (T e);  
    public void pop ();  
    public T top ();  
}
```

```
public class SimpleStack<T> implements IStack<T> {  
  
    private IStackImpl<T> impl;  
    // MyArrayList or MyLinkedList  
  
    public SimpleStack(IStackImpl<T> impl) {  
        this.impl = impl;  
    }  
  
    public void pop()           { impl.remove();           }  
    public void push(T e)       { impl.insert(e);       }  
    public T top()              { return impl.get(0); }  
}
```

```

public class LastModifiedStack<T> extends SimpleStack<T> {

    private Date lastModified;

    public LastModifiedStack(IStackImpl<T> impl) {
        super(impl);
        lastModified = new Date();
    }

    /** Push element and update date */
    public void push(T e) {
        lastModified = new Date();
        super.push(e);
    }

    /** Remove top element and update date */
    public void pop() {
        lastModified = new Date();
        super.pop();
    }

    public Date getLastModified() {
        return lastModified;
    }
}

```

LastModifiedStack אדישה למימוש של המחסנית, ותעבוד בצורה זהה עם כל מימוש שהוא


```
public interface IStackImpl<T> {  
    public void insert(T e);  
    public void remove();  
    public T get(int index);  
}
```

■ נושים לב להבדל שבין המונשק `IStack` ובין המונשק `IStackImpl`

■ המונשק `IStack` מייצג את המחסנית

■ המונשק `IStackImpl` מייצג את מימוש המחסנית

■ המחלקה `SimpleStack` המממשת את `IStack` מכילה מופע של מחלקה המממשת את `IStackImpl`

■ הורשה (מימוש) לצורכי מימוש (ייצוג) תתבצע מ `IStackImpl`

■ הורשה (מימוש) הנוגעת להפשטה תתבצע מ `IStack`

דוגמא למימוש המחסנית בArrayList

```
public class ArrayListStackImpl<E> implements IStackImpl<E> {  
    ArrayList<E> rep = new ArrayList<E>();  
  
    public E get(int index) { return rep.get(index); }  
    public void insert(E e) { rep.add(e); }  
    public void remove() { rep.remove(rep.size()-1); }  
}
```

איך יראה לקוח טיפוסי שמעוניין ליצור מופע של מחסנית?

```
SimpleStack<Integer> stack =  
    new SimpleStack<Integer> (new ArrayListStackImpl<Integer>());
```

- מה החסרונות של מבנה זה?
- איך ניתן לפתור אותם?

יש פה באג מורכב. המימוש של top
ב SimpleStack לא קונסיסטנטי
עם המימוש של remove

תבנית העיצוב Bridge

- אז מה יש לנו עד כה?
- שני מנשקים שמאפשרים לנו לייצר כל שילוב בין התנהגות למימוש.
- הגדרת המנשק `IStackImpl` מעט מלאכותית, ואף מאפשרת באגים מכיוון שאנחנו מאפשרים למשתמש לגשת למיקומים.
- נראה שהיה נכון להגדיר ב `IStackImpl` בדיוק את אותם השירותים שיש ב `IStack`.
- מצד שני – אנחנו רוצים לשמור על שני מנשקים שונים עצמאיים. כל מחסנית צריכה להיות הרכבה של `IStack` עם `IStackImpl`

עיצוב נוסף

```
public interface IStackBase<T>{  
    public void push (T e);  
    public void pop ();  
    public T top ();  
}
```

מתאר התנהגות (LastModified ,Bounded)

```
public interface IStack<T> extends IStackBase<T>{  
  
}
```

מתאר מימוש (ArrayList, LinkedList)

```
public interface IStackImpl<T> extends IStackBase<T>{  
  
}
```

עיצוב נוסף

```
public class SimpleStack<T> implements IStack<T> {  
  
    private IStackImpl<T> impl;  
    // MyArrayList or MyLinkedList  
  
    public SimpleStack(IStackImpl<T> impl) {  
        this.impl = impl;  
    }  
  
    public void pop()           { impl.pop();           }  
    public void push(T e)      { impl.push(e);      }  
    public T top()             { return impl.top(); }  
}
```

עיצוב נוסף

```
public class ArrayListStackImpl<E>
    implements IStackImpl<E> {
    ArrayList<E> rep = new ArrayList<E>();

    public E top()    { return rep.get(rep.size()-1);    }
    public void push(E e)  { rep.add(e);                }
    public void pop()     { rep.remove(rep.size()-1);    }
}
```

עיצוב נוסף

- בעיצוב החדש אנחנו שומרים על כך ש:
 - כל מחסנית היא הרכבה של התנהגות (IStack) ושל מימוש (IStackImpl)
 - מימוש יציב יותר – פחות פתח לבאגים בשונה מהעיצוב הקודם של IStackImpl
 - תודות להכמסה טובה יותר של IStackImpl
- האם מימשנו ירושה מרובה?
 - לא! אמנם אנחנו עושים שימוש חוזר בקוד של שתי מחלקות, אחת להתנהגות ואחת למימוש, כל מחסנית שנגדיר יורשת רק ממחלקה אחת.

עיצוב נוסף

```
SimpleStack<Integer> stack =  
    new SimpleStack<Integer>(new ArrayListStackImpl<Integer>());
```

- המחסנית stack מקיימת יחס is-a רק עם SimpleStack.
- למעשה, אין שום דרך לדעת שהמימוש הפנימי הוא ArrayListStackImpl.
- ואם זה מאוד חשוב לנו?

עיצוב נוסף

```
public interface IArrayListImpl{  
  
}
```

■ נוסף מנשק חדש שתפקידו לציין שהמחלקה ממומשת
באמצעות ArrayList

```
public class SimpleArrayListStack<T> extends SimpleStack<T>  
    implements IArrayListImpl{  
  
    public SimpleArrayListStack() {  
        super(new ArrayListStackImpl<>());  
    }  
  
}
```

עיצוב חדש

■ חסרונות:

- ריבוי מחלקות קונקרטיות: עבור כל הרכבה של מימוש והתנהגות נצטרך להגדיר מחלקה משלו.
- ריבוי מנשקים: לכל מימוש נצטרך להגדיר מנשק ריק משלו.

■ יתרונות:

- מחסנית הממומשת באמצעות `ArrayList`, ללא תלות בהתנהגות, תקיים יחס `is-a` עם `ArrayListImpl`
- ניתן למשל לשלוח אותה לפונק' שמטפלת במחסניות הממומשות באמצעות `ArrayList`

האם קיבלנו ירושה מרובה?

- קרוב, אבל לא.
- המחסנית SimpleArrayListStack מקיימת יחס is-a עם:
 - המחלקה SimpleStack
 - עם המנשק ArrayListImpl
- אם היתה ירושה מרובה אמיתית, היה מתקיים יחס is-a עם ArrayListStackImpl

קבלנות משנה -
על הורשה, טענות וחוזים

הורשה וטענות (assertions)

- תנאי קדם, תנאי בתר ושמורות שהוגדרו עבור מחלקה או מנשק תקפים גם לגבי צאצאי המחלקה (וממשי המנשק), ועשויים להשתנות
- עצם ממחלקה נגזרת המוצבע ע"י הפנייה מטיפוס המנשק [או טיפוס מחלקת הבסיס], צריך לקיים את שמורת המנשק [מחלקת הבסיס]
- מכאן ששמורה של כל מחלקה צריכה להיות שווה או חזקה יותר משמורת הוריה
- בגלל מנגנון הפולימורפיזם, אי הקפדה על כלל זה עשויה ליצור בעיות במערכת התוכנה, כפי שנדגים מיד



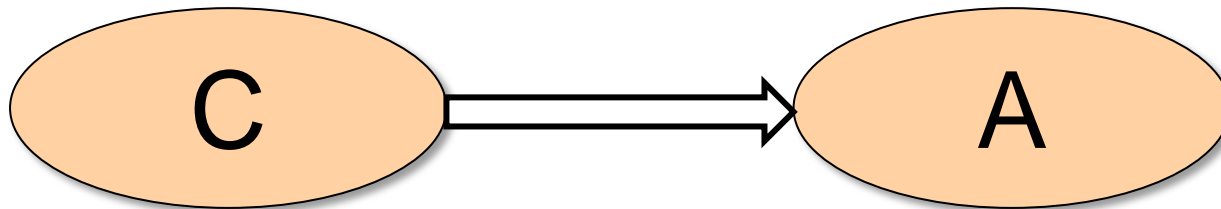
קבלנות משנה

■ מחלקת C היא לקוחה של מחלקה A, כלומר:

■ יש ל-C הפנייה ל-A (אחד השדות)

או

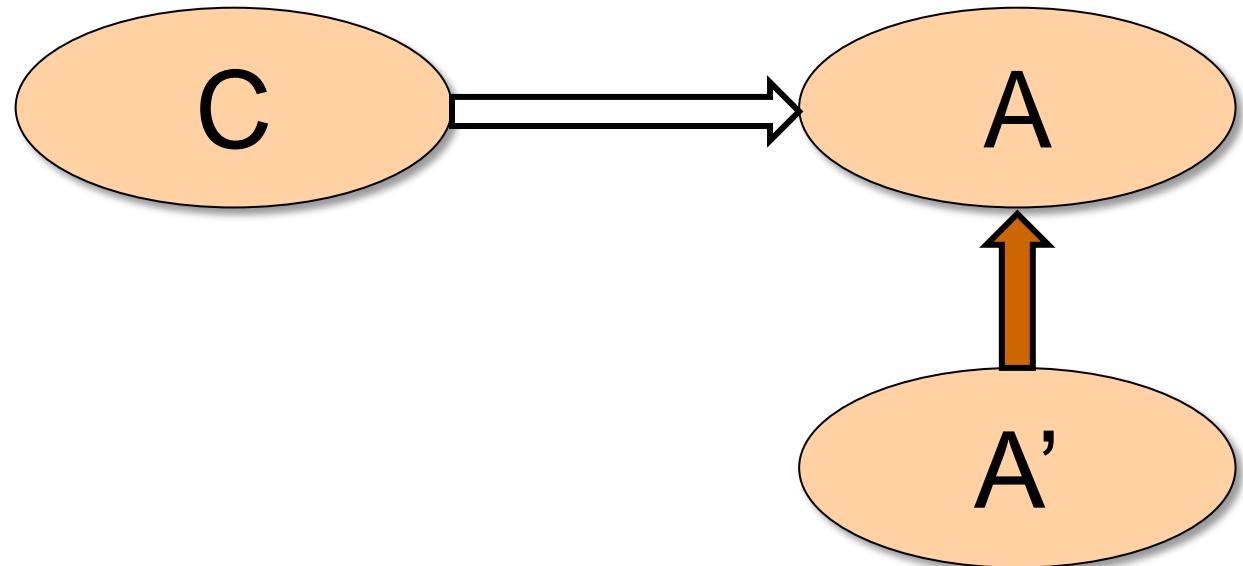
■ אחת המתודות של C מקבלת פרמטר מטיפוס A (הפנייה ל A)



■ C מכירה את השמורה של A ומצפה מ A לקיים אותה

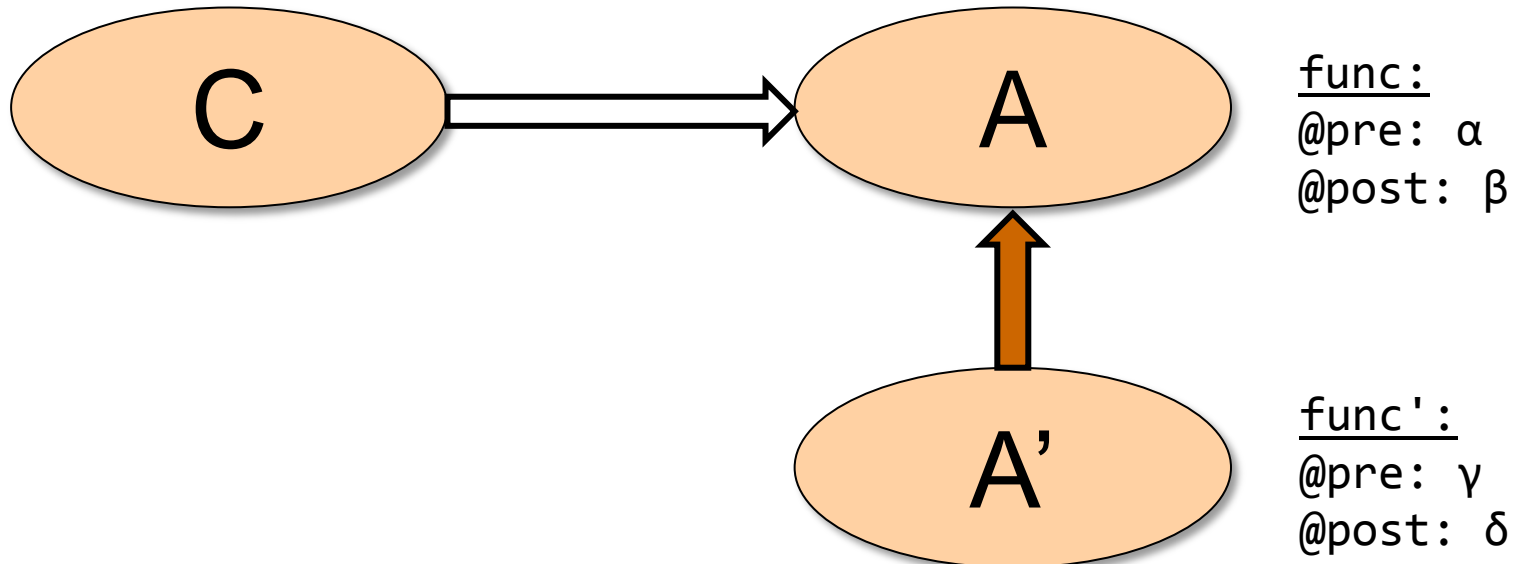
קבלנות משנה - השמורה

- בפועל, המצביע ל- A מצביע ל- A' , מחלקה הנורשת מ- A
- ברור שכדי לקיים יחסים פולימורפים תקינים על A' לקיים לפחות את שמורת A



קבלנות משנה – תנאי קדם ובתר

- המחלקה A' דורסת (overrides) שירות r() של A
- מה יש לדרוש מתנאי הקדם והבתר של השירות החדש ביחס לאלו של השירות המקורי?

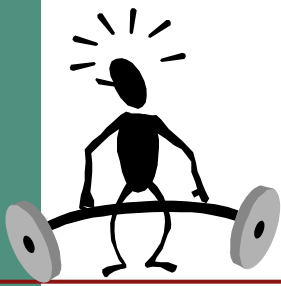


דוגמא

■ בתוך המחלקה Client מופיע הקוד הבא:

```
public class Client {  
    ...  
    public static void g(String[] args)  
    {  
        List<String> lst = Arrays.asList(args);  
        ...  
    }  
}
```

- בדוגמא זו Client הוא הלקוח (C) ו-List הוא הספק (A)
- ואולם ברור ש - lst מצביע בפועל לעצם ממחלקה שמממשת את List (אולי ArrayList). מחלקה זו היא קבלנית משנה (A')
- הלקוח, שאינו מכיר את קבלן המשנה שלו, מצפה ממנו לעמוד בחוזה המקורי (החוזה מול הספק)



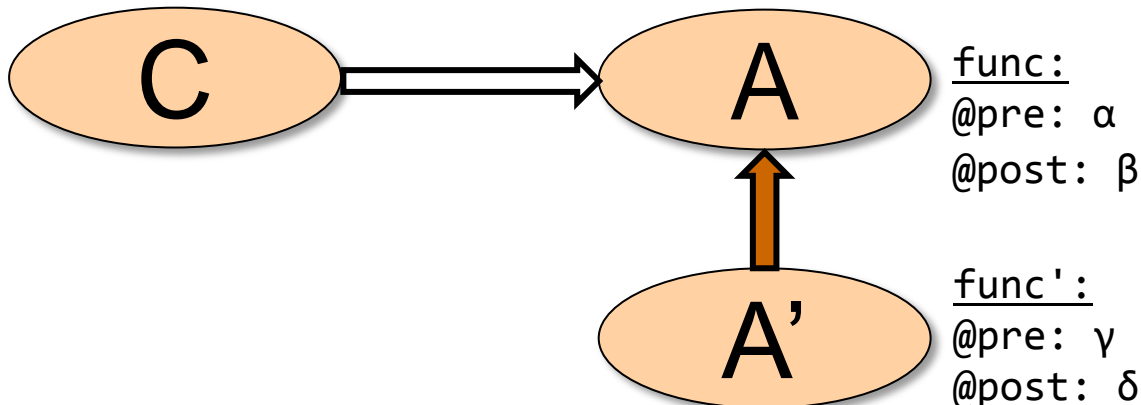
קבלנות משנה – תנאי קדם

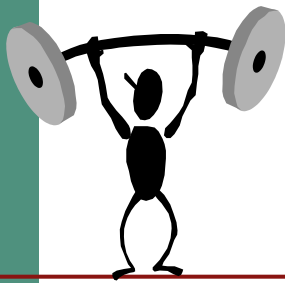
■ נניח כי במחלקה C מופיע הקוד הבא:

```
A aObj = ...;  
aObj.func();
```

■ על C לקיים את תנאי הקדם של $A.r()$: היא כלל אינה מכירה את המחלקה A' ואינה יודעת על קיום $A'.r()$

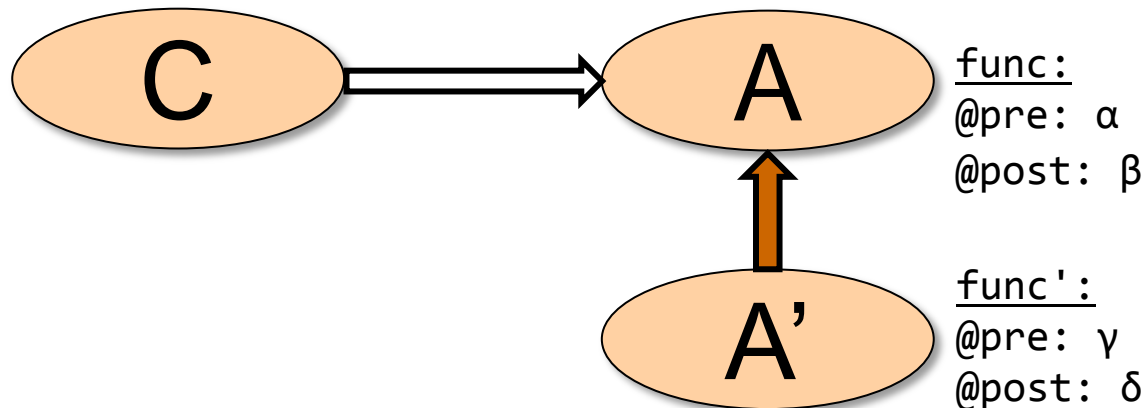
■ לכן על תנאי הקדם המוגדר במחלקה הנגזרת להיות שווה או חלש יותר מתנאי הקדם המקורי





קבלנות משנה – תנאי בתר

- משיקולים דומים על תנאי הבתר של המחלקה הנגזרת להיות שווה או חזק יותר מתנאי הבתר המקורי
- ללקוח C 'הובטח' β ע"י A ואסור שמאחורי הקלעים יסופק δ החלש ממנו
- מנגנון זה מכונה "קבלנות משנה" (subcontracting)



הטענות האפקטיביות

- השמורה ה'אמיתית' של מחלקה מורכבת מ **AND** לוגי של כל הטענות המופיעות בשמורת אותה מחלקה ובכל הוריה לאורך עץ ההורשה
- תנאי הקדם ה'אמיתי' של מתודה שהוגדרה מחדש במחלקה כלשהי, הוא ה **OR** הלוגי של כל תנאי הקדם של מתודה זו בכל הוריה של אותה מחלקה לאורך עץ ההורשה
- תנאי הבתר ה'אמיתי' של מתודה שהוגדרה מחדש במחלקה כלשהי הוא ה **AND** הלוגי של כל תנאי הבתר האפקטיביים של מתודה זו בכל הוריה של אותה מחלקה לאורך עץ ההורשה

דוגמא

```
public class MathWizard {  
    ...  
    /** returns the square root of num  
     * @pre epsilon >= 10 ^ (-6)  
     * @post abs($ret*$ret - num) <= epsilon  
     */  
    double sqrt(int num, double epsilon);  
    ...  
}
```

דוגמא

```
public class AccurateMathWizard extends MathWizard {  
    ...  
    /** returns the square root of num  
     * @pre epsilon >= 10-20  
     * @post abs($ret*$ret - num) <= epsilon/2  
     */  
    double sqrt(int num, double epsilon);  
    ...  
}
```

בדוגמא תנאי הקדם חלש יותר (מרשה יותר ערכי אפסילון) ■
ותנאי הבתר יותר חזק (מבטיח דיוק רב יותר)

קבלנות משנה

- משהבנו את ההיגיון שבבסיס יחסי ספק, לקוח וקבלן משנה, ניתן להסביר את חוקי שפת Java לגבי השינויים הבאים שקבלן המשנה יכול לבצע:
 - שינוי ההצהרה על חריגים
 - שינוי נראות
 - שינוי הערך המוחזר

הורשה וחריגים

קבלן משנה (מחלקה יורשת [מממשת], הדורסת [מממשת] שרות) אינו יכול לזרוק מאחורי הקלעים חריג שלא הוגדר בשרות הנדרס [או במנשק]

למתודה הדורסת [המממשת] **מותר להקל** על הלקוח ולזרוק פחות חריגים מהמתודה במחלקת הבסיס שלה [במנשק]

לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שיורשת מ A?

```
public class A{  
    public void func() throws IOException{ }  
}
```

```
public class B extends A{  
    ✓ //public void func() {}  
    ✓ //public void func() throws IOException {}  
    ✓ //public void func() throws EOFException{}  
    ✗ //public void func() throws Exception{}  
}
```


הורשה וניראות

■ למתודה הדורסת [המממשת] **מותר להקל** את הנראות – כלומר להגדיר סטטוס נראות רחב יותר, אבל אסור להגדיר סטטוס נראות מצומצם יותר.

■ לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שירשת מ A?

```
public class A{  
    protected void func(){ }  
}
```

```
public class B extends A{  
    ✓ //public void func(){ }  
    ✓ //protected void func() {}  
    ✗ //void func() {}  
    ✗ //private void func() {}  
}
```

הורשה והערך המוחזר

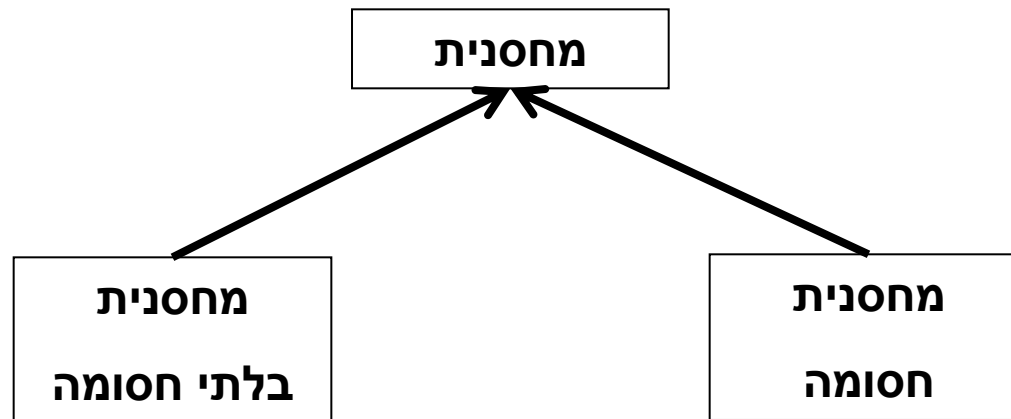
- למתודה הדורסת [המממשת] **מותר לצמצם** את טיפוס הערך המוחזר, כלומר טיפוס הערך המוחזר הוא תת טיפוס של טיפוס הערך המוחזר במתודה במחלקת הבסיס שלה [במנשק]
- לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שירשת מ A?

```
public class A{  
    public Number func() { return null; }  
}
```

```
public class B extends A{  
    ✗ //public Object func() { return null; }  
    ✓ //public Number func() { return null; }  
    ✓ //public Integer func() { return null; }  
}
```

תנאי קדם מופשט

- מהי ההיררכיה בין 3 המחלקות: מחסנית, מחסנית חסומה, מחסנית בלתי חסומה?



- מה יהיה תנאי הקדם של המתודה `push` במחלקה מחסנית?

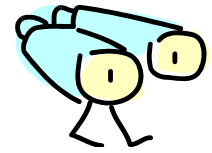
תנאי קדם מופשט

- תנאי הקדם לא יכול להיות ריק (TRUE) כי אז הוא יחוזק ע"י המחסנית החסומה
- תנאי הקדם צריך להיות `!full()` כאשר `full()` היא מתודה מופשטת (או מתודה המחזירה תמיד `false`). המחלקה מחסנית חסומה" תממש אותה כך שתחזיר `count() == capacity()`
- תנאי קדם המכיל מתודות מופשטות או מתודות שנדרסות במורד עץ ההורשה נקרא **תנאי קדם מופשט**
- למרות שתנאי הקדם הקונקרטי אכן מתחזק ע"י המחסנית החסומה תנאי הקדם המופשט נשאר ללא שינוי

תנאי קדם מופשט

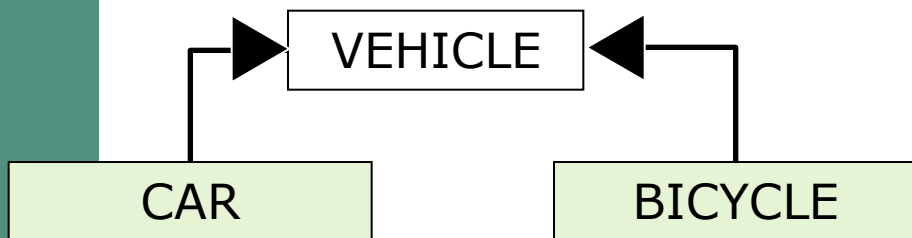
- כאשר מחלקת הבסיס מופשטת, תנאי קדם טריוויאליים מחייבים לפעמים **ראייה לעתיד**, כדי שלא יחזקו במחלקות נגזרות
- ראייה לעתיד אינה דבר מופרך במחלקות מופשטות
- נתבונן בדוגמא נוספת: מערכת תוכנה אשר מיוצגים בה כלי תחבורה שונים כגון מכונית, אווירון ואופניים

ראייה לטווח רחוק



- האבולוציה של היררכית מחלקות כלי הרכב לא מתחילה בגזירת מחלקות קונקרטיות שיירשו מ VEHICLE
- הגיוני יותר שבמהלך מימוש ו\או עיצוב המחלקות CAR ו- AIRPLANE נגלה שיש להן הרבה מן המשותף, וכדי למנוע שכפול קוד ניצור מחלקה שלישית - VEHICLE שתכיל את החיתוך של שתיהן
- אף כלי רכב אינו רק VEHICLE
- בראייה זו, אין זה מוגזם לדרוש ממחלקה מופשטת ניסוח תנאי קדם מופשט

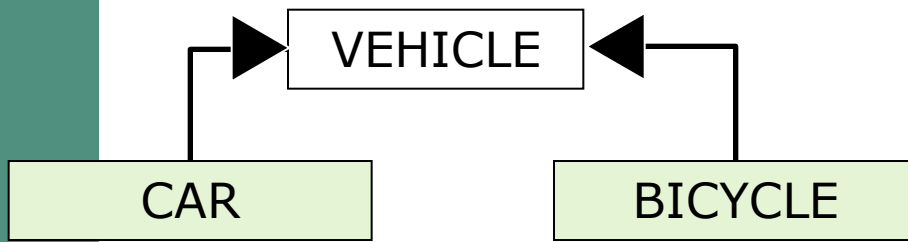
דוגמא



- מהו תנאי הקדם של המתודה `go()` של המחלקה `VEHICLE` ?
- על פניו – אין כל תנאי קדם לפעולה מופשטת
- מה עם המחלקה `CAR` ? – לה בטח יש דרישות כגון `hasFuel()`
- מה עם המחלקה `BICYCLE` ? – לה בטח יש דרישות כגון `hasAir()`
- איך `VEHICLE` תגדיר תנאי קדם ל `go()` גם כללי מספיק וגם שלא יחוזק ע"י אף אחד מירשותיה?



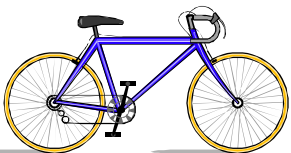
פתרון



- מתודה בולאנית כגון `canGo()` תעשה את העבודה

- המתודה תוגדר כמחזירה `TRUE` עבור `VEHICLE` (או שתוגדר כ `abstract`), ועבור כל אחת מירשותיה תידרס ותוגדר לפי מה שמתאים במחלקה האמורה

- בעצם המתודה `go()` היתה צריכה להיקרא `"go_because_you_can()"` וכך לא היתה כל הפתעה בתנאי הקדם "המוזר"



הורשה זה רע?

- הורשה היא מנגנון אשר חוסך קוד ספק
- פרט למנגנון הרב-צורתיות (polymorphism) הורשה היא סוכר תחבירי של הכלה ואינה הכרחית
 - במקום ש B יירש מ- A , ל- B יכולה להיות התכונה A (שדה)
- יחסי הורשה נכונים הם דבר עדין
 - יחס is-a לעומת יחס has-a או is-part-of
 - לעומת זאת To be is also to have אבל לא להיפך (משאית היא מכונית כלומר חלק בה הוא מכונית)
- לפעמים נוח לשאול "האם יכולים להיות לו שניים?"
 - לדוגמא: למכונית יש מנוע, האם יכולה להיות מכונית עם שני מנועים
- הורשה או מופע?
- האם Washington יורשת מ- State?



הכוח משחית

■ על המחלקה היורשת לקיים את 2 העקרונות:

■ יחס is-a

■ עקרון ההחלפה

■ אי שמירה על כך תגרום לעיוותים במערכת התוכנה

■ לדוגמא: ננסה לבטא את יחס המחלקות Rectangle ו-Square בעזרת הורשה

לא מתקיים יחס is-a

מלבן לא יורש מריבוע

```
public class Square {  
    protected double length;  
  
    public double getLength() {  
        return length;  
    }  
  
    public double getWidth() {  
        return length;  
    }  
  
    public double area() {  
        return length*length;  
    }  
    ...  
}
```

```
public class Rectangle  
    extends Square {  
    protected double width;  
  
    public double getWidth() {  
        return width;  
    }  
  
    public double area() {  
        return length*width;  
    }  
    ...  
}
```

Rectangle is **NOT** a Square – ברור כי העיצוב לקוי

למשל **המשתמר** של Square צריך להכיל את `getLength() == getWidth()` ■
וברור כי **Rectangle** לא שומר על כך ■

לא מתקיים
עקרון ההחלפה!

אז אולי ריבוע יורש ממלבן?

```
public class Rectangle {  
    protected double width;  
    protected double length;
```

■ מתקיים יחס is-a (ריבוע הוא מלבן) אבל
במימוש הספציפי הזה לא מתקיים עקרון
ההחלפה.

```
    public double getWidth() {  
        return width;  
    }
```

■ לא ניתן להשתמש בריבוע בכל הקשר שבו ניתן
היה להשתמש במלבן

```
    public double getLength() {  
        return length;  
    }
```

■ זה מפתיע – מכיוון שמתמטית ריבוע הוא סוג
של מלבן

```
    public double area() {  
        return length*width;  
    }
```

■ אז איך בכל זאת נממש את המחלקות ריבוע
ומלבן?

■ בעולם התוכנה יש לעשות "ויתורים כואבים"

```
public static void widen(Rectangle rect, double delta) {  
    rect.width += delta;  
}
```

```
...  
}
```