

תוכנה 1 בשפת Java  
שיעור מספר 13: "המשך הורשה, אוספים  
גנריים וסיכום"

# היום בשיעור

- קבלנות משנה (הורשה והחזקה) – המשך משיעור 12.
- המשך מוזרויות ב Java
- אוספים גנריים
- בחינה

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



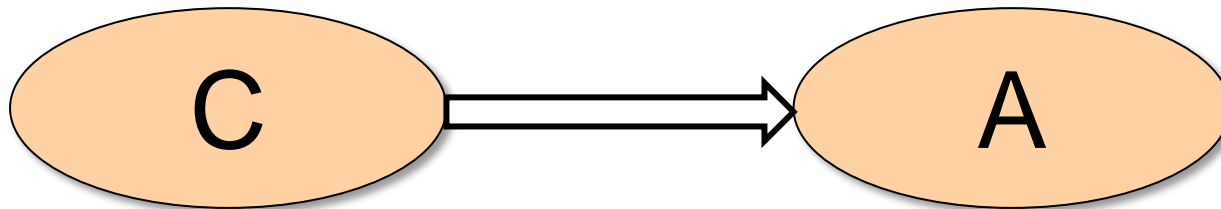
# קבלנות משנה

■ מחלקת C היא לקוחה של מחלקה A, כלומר:

■ יש ל-C הפנייה ל-A (אחד השדות)

או

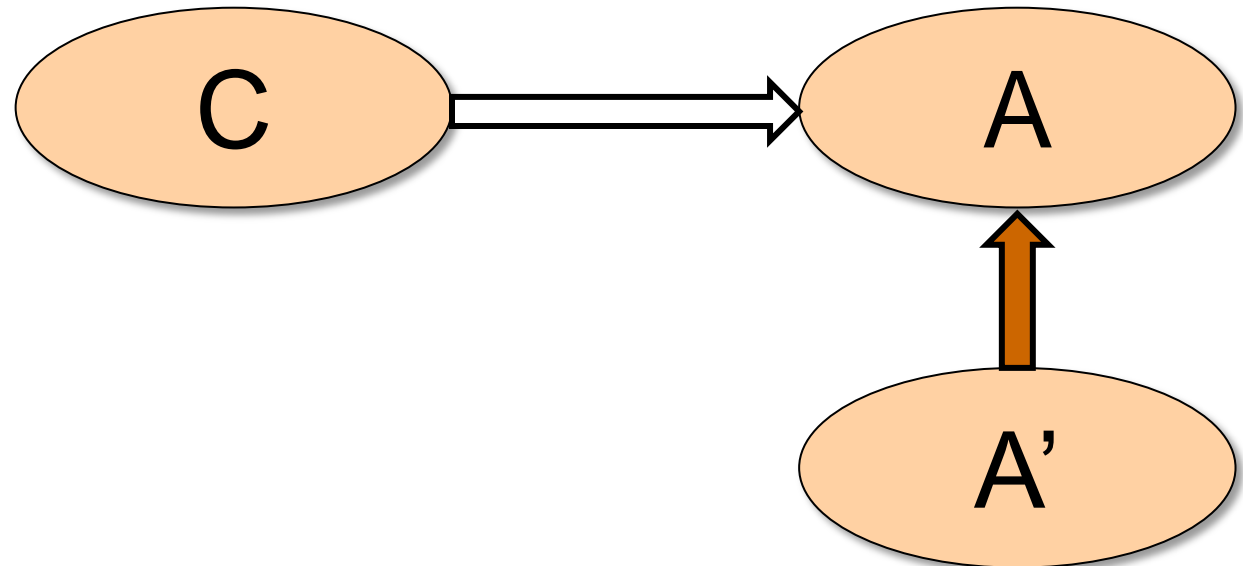
■ אחת המתודות של C מקבלת פרמטר מטיפוס A (הפנייה ל A)



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

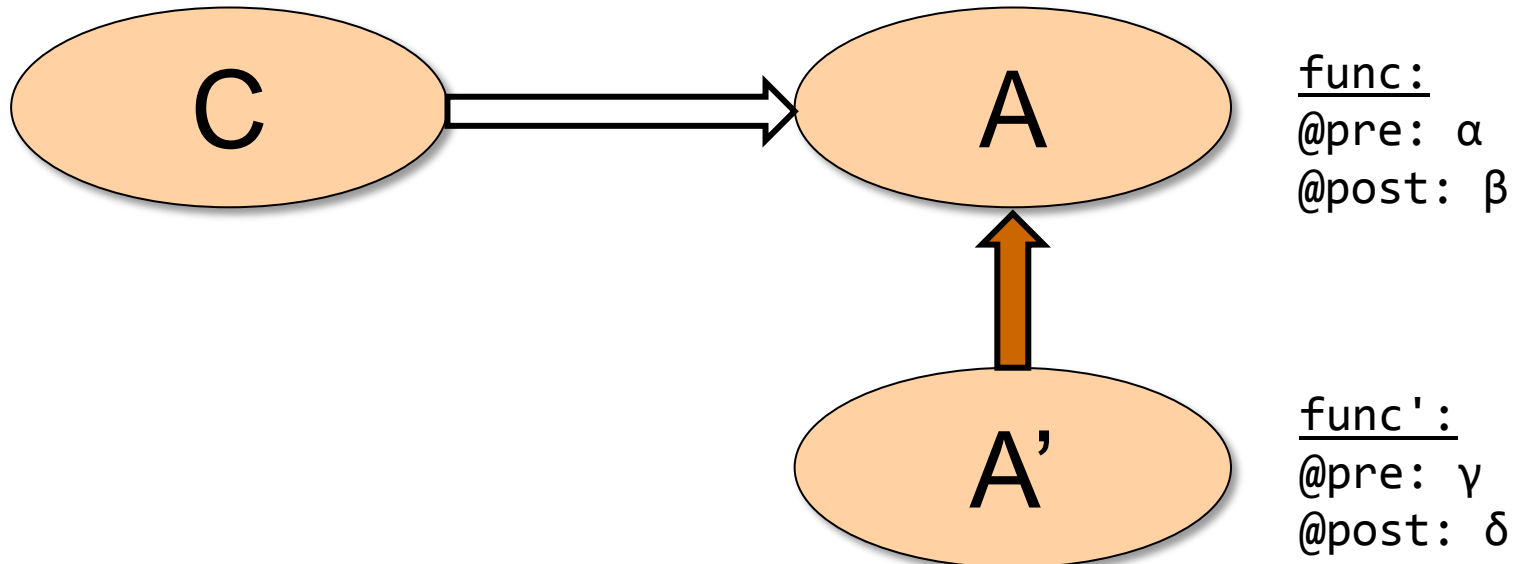
# קבלנות משנה - השמורה

- 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?



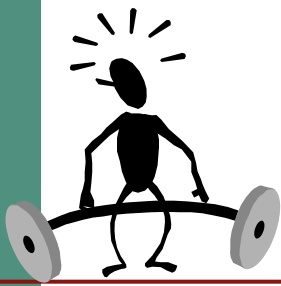
# דוגמא

■ 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)





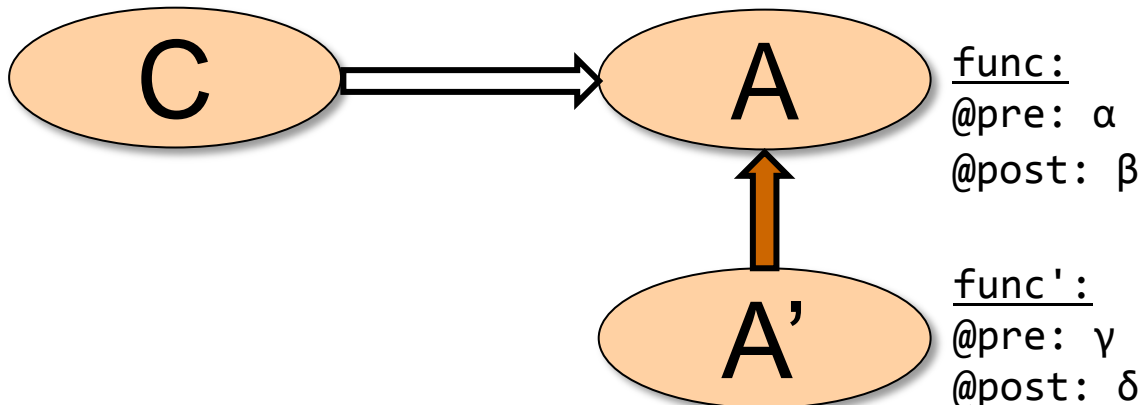
# קבלנות משנה – תנאי קדם

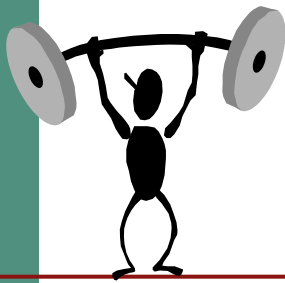
■ נניח כי במחלקה C מופיע הקוד הבא:

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

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

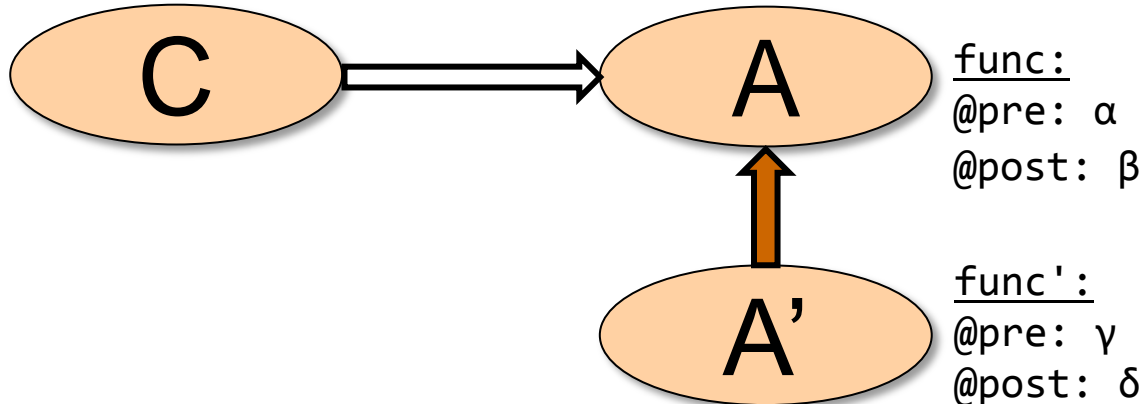
■ לכן על תנאי הקדם המוגדר במחלקה הנגזרת להיות שווה או חלש יותר מתנאי הקדם המקורי





# קבלנות משנה – תנאי בתר

- משיקולים דומים על תנאי הבתר של המחלקה הנגזרת להיות שווה או חזק יותר מתנאי הבתר המקורי
- ללקוח C 'הובטח'  $\beta$  ע"י A ואסור שמאחורי הקלעים יסופק  $\delta$  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 שיורשת 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

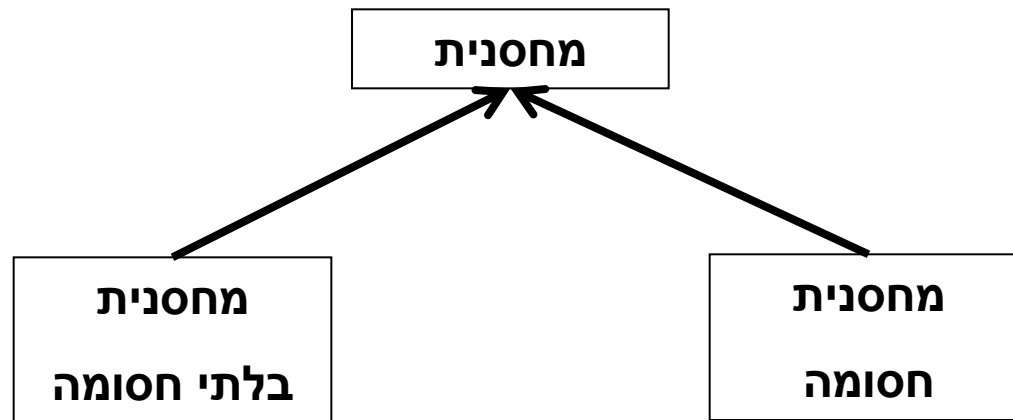
- למתודה הדורסת [המממשת] **מותר לצמצם** את טיפוס הערך המוחזר, כלומר טיפוס הערך המוחזר הוא תת טיפוס של טיפוס הערך המוחזר במתודה במחלקת הבסיס שלה [במנשק]
- לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שירשת מ A?

```
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

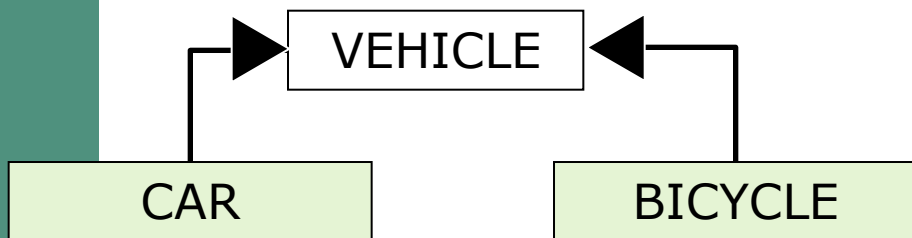
- כאשר מחלקת הבסיס מופשטת, תנאי קדם טריוויאליים מחייבים לפעמים **ראייה לעתיד**, כדי שלא יחזקו במחלקות נגזרות
- ראייה לעתיד אינה דבר מופרך במחלקות מופשטות
- נתבונן בדוגמא נוספת: מערכת תוכנה אשר מיוצגים בה כלי תחבורה שונים כגון מכונית, אווירון ואופניים

# 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

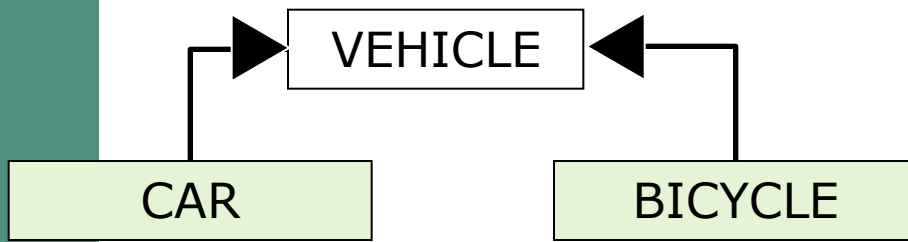
# דוגמא



- 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?



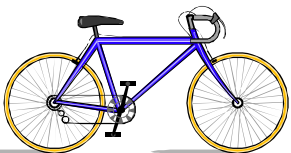
# פתרון



- 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;  
}
```

...

}

# מוזריות ב Java

```
public class Base {
    private void priv() { System.out.println("priv in Base"); }
    public void pub() { System.out.println("pub in Base"); }

    public void foo() {
        priv();
        pub();
    }
}
```

```
public class Sub extends Base {
    private void priv() { System.out.println("priv in Sub"); }
    public void pub() { System.out.println("pub in Sub"); }
}
```

```
public class Test {

    public static void main(String[] args) {
        Base b = new Sub();
        b.foo();
    }
}
```

מה יודפס?

priv in Base  
pub in Sub

# שדות, הורשה וקישור סטטי

- גם קומפילציה של התייחסויות לשדות מתבצעת בצורה סטטית
- מחלקה יורשת יכולה להגדיר שדה גם אם שדה בשם זה היה קיים במחלקת הבסיס (מאותו טיפוס או טיפוס אחר)

```
public class Base {  
    public int i = 5;  
}
```

```
public class Sub extends Base {  
    public String i = "five";  
}
```

```
5  
five  
5
```

```
public class Test {  
  
    public static void main(String[] args) {  
        Base bb = new Base();  
        Sub ss = new Sub();  
        Base bs = new Sub();  
  
        System.out.println(bb.i);  
        System.out.println(ss.i);  
        System.out.println(bs.i);  
    }  
}
```

מה יודפס?

# העמסה והורשה

■ במקרים של העמסה הקומפיילר מחליט איזו גרסה תרוץ (יותר נכון: איזו גרסה לא תרוץ)

■ זה נראה סביר (הפרוצדורות מתוך `java.lang.String`):

```
static String valueOf(double d)      {...}  
static String valueOf(boolean b)    {...}
```

■ אבל מה עם זה?

```
overloaded(Rectangle      x) {...}  
overloaded(ColoredRectangle x) {...}
```

■ לא נורא, הקומפיילר יכול להחליט,

```
Rectangle      r = new ColoredRectangle ();  
ColoredRectangle cr = new ColoredRectangle ();  
overloaded(r); // we must use the more general method  
overloaded(cr); // The more specific method applies
```

# העמסה והורשה

■ אבל זה כבר מוגזם:

```
overTheTop(Rectangle x, ColoredRectangle y) {...}  
overTheTop(ColoredRectangle x, Rectangle y) {...}
```

```
ColoredRectangle a = new ColoredRectangle ();  
ColoredRectangle b = new ColoredRectangle ();  
overTheTop(a, b);
```

- ברור שנדרשת המרה (casting) אבל של איזה פרמטר? a או b?
- אין דרך להחליט; הפעלת השגרה לא חוקית בג'אווה



# העמסה והורשה - שבריריות

```
overTheTop(Rectangle x, ColoredRectangle y) {...}  
overTheTop(ColoredRectangle x, Rectangle y) {...}
```

```
ColoredRectangle a = new ColoredRectangle ();  
ColoredRectangle b = new ColoredRectangle ();  
overTheTop(a, b);
```

- אם הייתה רק הגרסה הירוקה, הקריאה לשגרה הייתה חוקית
- כאשר מוסיפים את הגרסה הסגולה, 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 שברירי

# העמסה והורשה - שבריריות

```
public class A {  
    /*  
    public void func(Object o, String s) {  
        System.out.println("calling version A");  
    }*/  
}  
  
public class B extends A{  
    public void func(String s, Object o) {  
        System.out.println("calling version B");  
    }  
}  
  
public class C {  
    public static void main(String[] args) {  
        B a = new B();  
        a.func("abc", "abc");  
    }  
}
```

אם נוציא את הקוד ב A מההערה ונקמפל רק את A, C תמשיך לרוץ עם הגירסא של B.

# העמסה והורשה - יותר גרוע

```
class B {
    overloaded(Rectangle      x) {...}
}

class S extends B {
    overloaded(Rectangle      x) {...} // override
    overloaded(ColoredRectangle x) {...} // overload
    but no override!
}

S o = new S();
ColoredRectangle cr = ...
o.overloaded( cr );           // invoke the purple
((B) o).overloaded( cr )    // What to invoke?
```

```

class B {
    overloaded(Rectangle x) {...}
}

class S extends B {
    overloaded(Rectangle x) {...} // override
    overloaded(ColoredRectangle x) {...} // overload but no override!
}

S o = new S();
ColoredRectangle cr = ...
o.overloaded( cr ); // invoke the purple
((B) o).overloaded( cr ) // What to invoke?

```

■ 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( cr )` תיבחר (בזמן קומפילציה) החתימה: `B.overloaded(Rectangle)`

■ 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( cr )`

# העמסה זה רע

- 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

# שאלות מהקהל וחזרה

```

public static int comparesCounter;
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(5,4,3,2,1,6);
    ints.stream()
        .filter(x->x%2==0)
        .peek(x->{System.out.println("peek " + x);})
        .sorted((x,y)->{
            comparesCounter++;
            System.out.println("comparing: " + x + "," + y);
            return Integer.compare(x, y);
        })
        .forEach(System.out::println);
    System.out.println("num of compares: " + comparesCounter);
}

```

```

output:
peek 4
peek 2
peek 6
comparing: 2,4
comparing: 6,2
comparing: 6,4
2
4
6
num of compares:
3

```

הפעולה peek מחזירה את הזרם עליו היא מופעלת, ובנוסף, מפעילה על כל איברי הזרם את הפעולה שקיבלה כפרמטר.

הפעולה sorted אינה פעולה שגרתית. על מנת לבצע אותה, יש לאסוף את כל אברי הזרם עליו היא מופעלת!

מה ישתנה אם נבצע את פעולת ה filter אחרי פעולת ה sort?

```

public class Test{
    public static void main(String[] args) {
        func("abc");
    }

    public static SimpleClass func(Object i) {
        try {
            String str = (String)i;
            return new SimpleClass(str);
        }
        finally {
            System.out.println("finally!");
        }
    }

    public static class SimpleClass{
        public SimpleClass(String str) {
            System.out.println("*: " + str);
        }
    }
}

```

האם בלוק ה `finally`  
יתבצע?  
כמובן!  
נשאלת השאלה: מתי?  
לפני שהמחשנית משחררת  
את הפונקציה. ערך  
ההחזרה נוצר לפני  
שמתבצע בלוק ה `finally`.



```
/**
 * @inv !isEmpty() implies top() != null
 */
public class SectionA {
```

```
    private final LinkedList<Object> elements = new LinkedList<>();
```

```
    /**
     * @post !isEmpty()
     * @post top() == o
     */
    public void push(Object o){
        elements.add(o);
    }
```

```
    /**
     * @pre !isEmpty()
     * @post @return == @prev(top())
     */
    public Object pop(){
        final Object popped = top();
        elements.removeLast();
        return popped;
    }
```

שאלה מתרגיל בית 6:  
אילו מבין הפונקציות מפרות את שמורת  
המחלקה?

הפונקציה push מפרה  
את השמורה.  
לאחר השימוש ב push  
יכול להיווצר מצב שבו  
top מחזירה null למרות  
שהמחסנית אינה ריקה

```

/**
 * @inv !isEmpty() implies top() != null
 */
public class SectionA {
    //previous methods
    /**
     * @pre !isEmpty()
     * @post @return != null
     */
    public Object top(){
        return elements.getLast();
    }

    /**
     *
     * @post @return == true iff elements.size() == 0
     */
    public boolean isEmpty(){
        return elements.size() == 0;
    }
}

```

הפונק' top תפר את השמורה רק אם יש null במחסנית. איך ה null הגיע לשם? אם push ממומשת נכון, זה לא יכול לקרות.

החוזה מתייחס למימוש הפנימי, לכן, תנאי ה post צריך להיות מומר ב post\_impl

# מחלקות פנימיות בתוך מתודות

- ניתן להגדיר מחלקה פנימית בתוך מתודה של המחלקה החיצונית

- הדבר מגביר את תחום ההכרה של אותה מחלקה לתחום אותה המתודה בלבד.

- המחלקה הפנימית תוכל להשתמש במשתנים מקומיים של המתודה רק אם הם הוגדרו כ `final`, החל מ Java 8 ניתן לגשת גם למשתנים ש"מתנהגים" כמו משתנים שהוגדרו כ `final`, כלומר, מקבלים השמה פעם אחת בלבד לאורך חייהם (`effectively final`).

# מחלקות פנימיות בתוך מתודות

```
public class Test {
    public void test(int num) {
        final int x = num+3;
        int y = num*2;
        int z = num-1;
        class Info{
            public String toString() {
                return "***" + x + "***" + y + "***" + z;
            }
        }
        z = 4;
        System.out.println(new Info());
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.test(5);
    }
}
```

האם toString יכולה  
לגשת גם ל num?

מה יקרה לקוד לאחר  
הוספת שורה זו?

# אוספים גנריים

# HashSet

```
class Point{
    int x;
    int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```
Set<Point> points = new
HashSet<>();
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
points.add(p1);
points.add(p2);
System.out.println(points.size());
```

Output:  
2

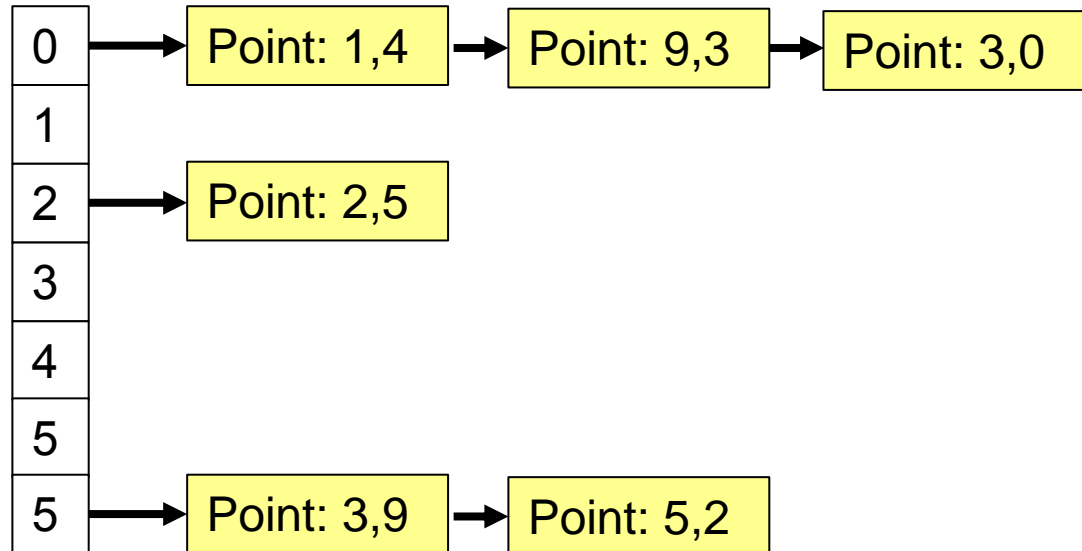


# HashSet

איך עובדת הכנסה ל HashSet ?

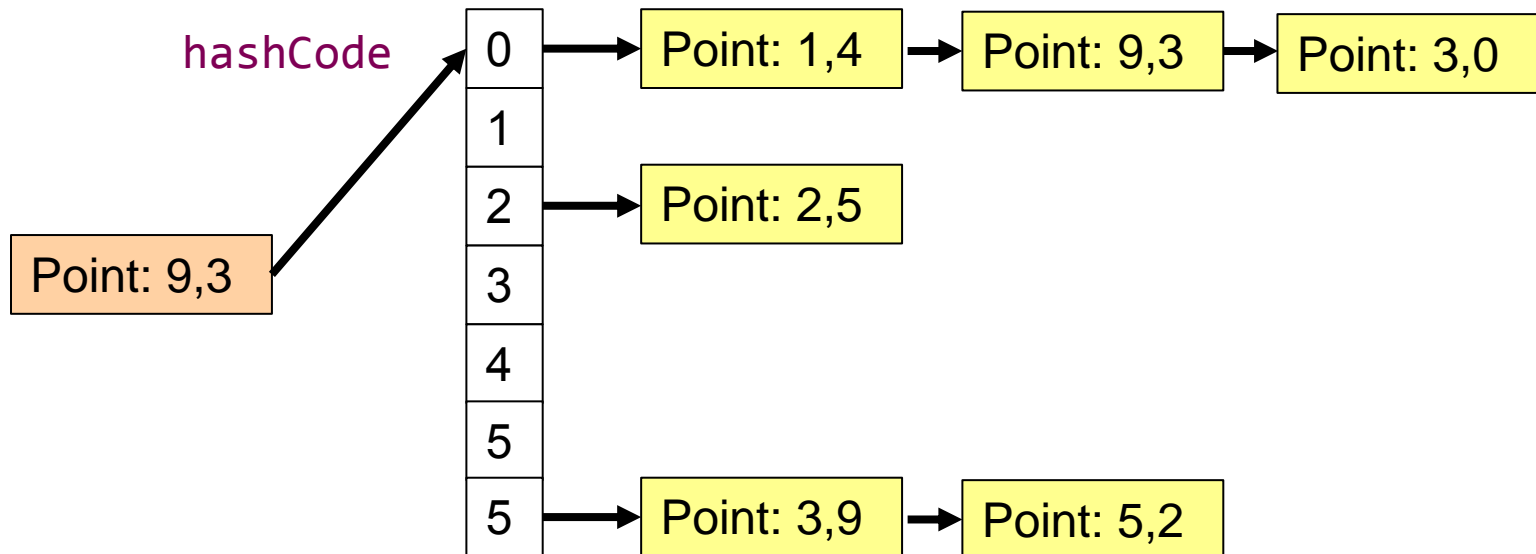
Point: 9,3

?



# HashSet

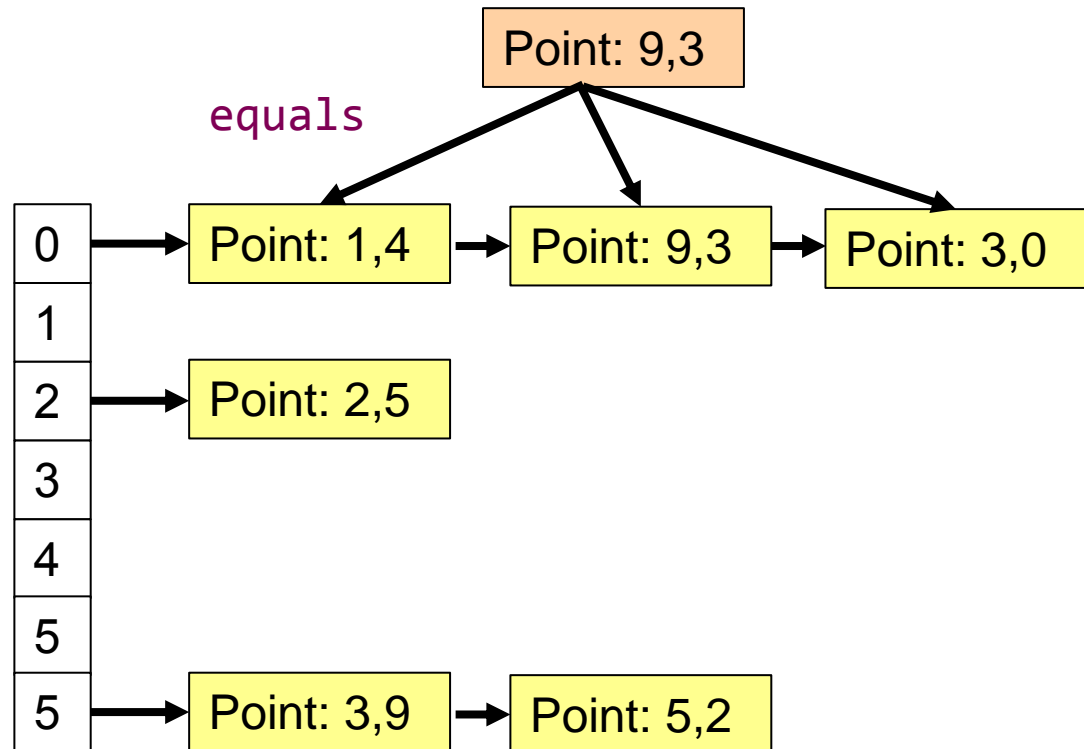
איך עובדת הכנסה ל HashSet?





# HashSet

איך עובדת הכנסה ל HashSet?



# HashSet

- דרישות מהמימוש של hashCode:
  - עבור אותו האובייקט, hashCode צריכה להחזיר את אותו הערך בכל קריאה.
  - אם שני אובייקטים  $x$  ו  $y$  מקיימים  $x.equals(y)$ , הפונקציה hashCode צריכה להחזיר את אותו הערך עבור שניהם
  - כדאי לייצר ערכים שונים עבור אובייקטים  $x$  ו  $y$  שאינם מקיימים  $x.equals(y)$  על מנת לשפר ביצועים.
  - ייצור ערכים זהים יפגע רק בביצועים, לא בנכונות.

# HashSet

- כדאי לתת ל eclipse לחולל לבד את המימוש של hashCode, ביחד עם המימוש של equals.
- צריך לוודא שמעדכנים את שני המימושים כאשר יש שינוי באובייקטים (למשל, מתווספים או מוסרים שדות).
- HashMap – עובד בדיוק באותו האופן.

# TreeSet

```
Set<Point> points = new TreeSet<>(
    (a,b)->Integer.compare(a.x, b.x));
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
points.add(p1);
points.add(p2);
System.out.println(points.size());
```

Output:

1

חייבים לשלוח  
Comparator כיוון ש  
Point אינה  
Comparable

# TreeSet

```
Set<Point> points = new TreeSet<>(
    (a,b)->Integer.compare(a.x, b.x));
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
Point p3 = new Point(1,3);
points.add(p1);
points.add(p2);
points.add(p3);
System.out.println(points.size());
```

Output:

1



# TreeSet

- TreeSet אינו עובד עם hashCode (הגיוני, בשביל זה יש HashSet).
- TreeSet אינו עובד עם equals (זה קצת מפתיע).
- המימוש של compare/compareTo (תלוי אם האלמנטים הם Comparable או שמתמשים ב Comparator) חייב להיות עקבי עם equals.
- אחרת – נוכל לגלות ששני אובייקטים שאינם equals נחשבים כזהים ע"י ה TreeSet.

# בחינה

- החומר לבחינה – כל החומר, כולל תרגולים ותרגילי בית
- בחינה אמריקאית
- פורום שאלות מבחינות קודמות
- ערעור קבוצתי
- מנגנון שאלת שאלות בזמן הבחינה.

תודה!