תוכנה 1 בשפת Java

שיעור מספר 13: "המשך הורשה, סיכום"

היום בשיעור

- קבלנות משנה
- העמסה וירושה
- תבנית העיצוב Bridge (אולי)
- בחינה

קבלנות משנה -
על הורשה, טענות וחוזים

הורשה וטענות (assertions)

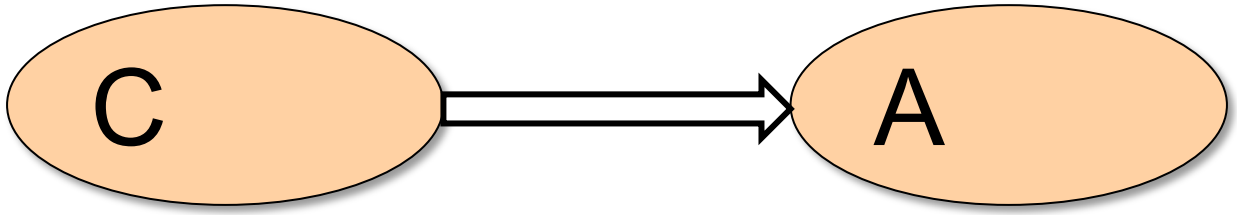
- תנאי קדם, תנאי בתר ושמורות שהוגדרו עבור מחלקה או מנשק תקפים גם לגבי צאצאי המחלקה (וממשי המנשק), ועשויים להשתנות
- עצם ממחלקה נגזרת המוצבע ע"י הפנייה מטיפוס המנשק [או טיפוס מחלקת הבסיס], צריך לקיים את שמורת המנשק [מחלקת הבסיס]
- מכאן ששמורה של כל מחלקה צריכה להיות שווה או חזקה יותר משמורת הוריה
- בגלל 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

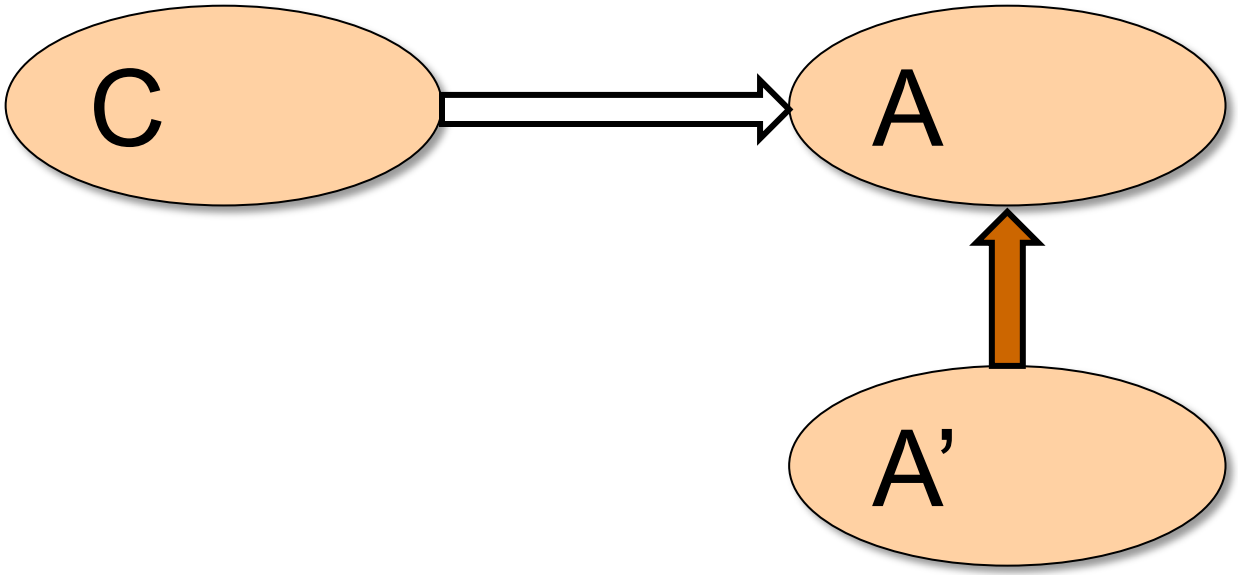
  - אחת המתודות של C מקבלת פרמטר מטיפוס A (הפנייה ל A)



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

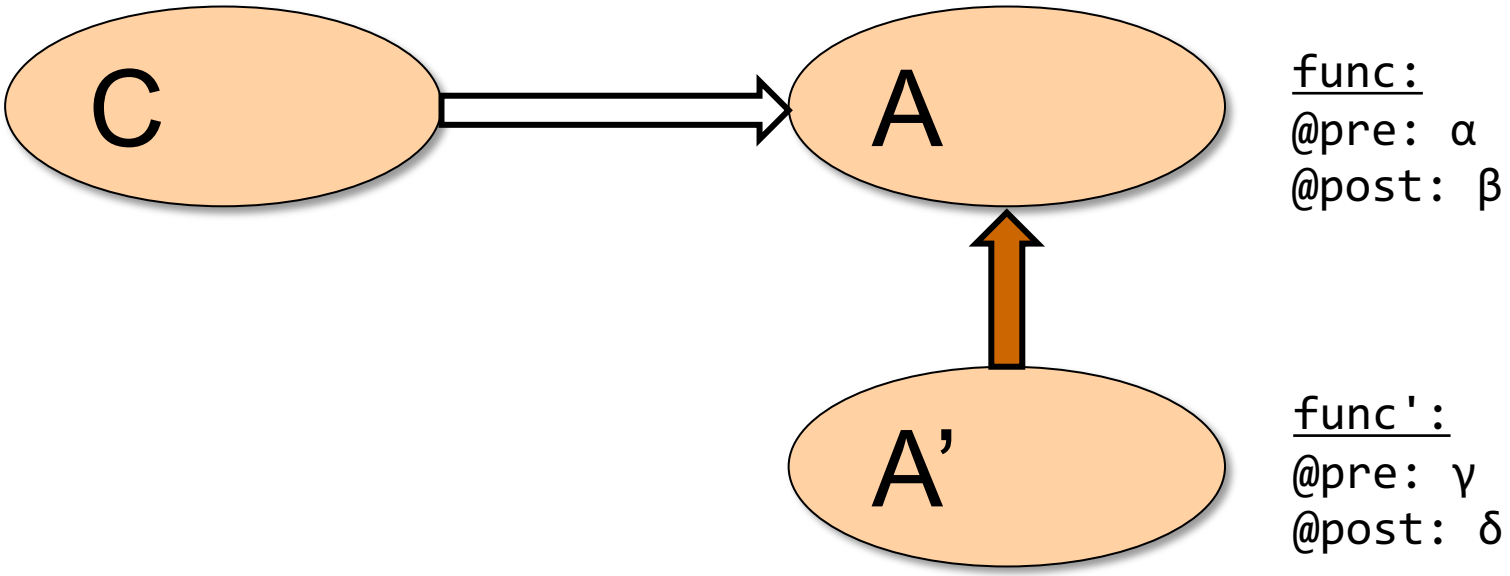
קבלנות משנה - השמורה

- 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?

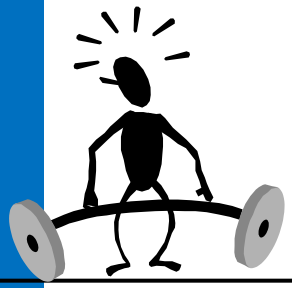


דוגמא

- 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 שממשת 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)

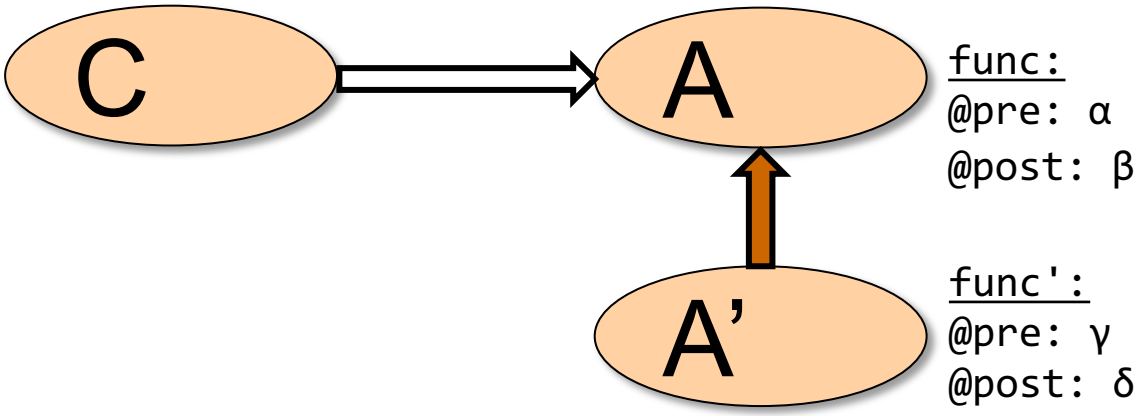


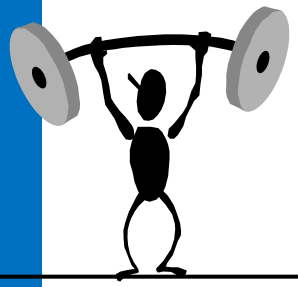
קבלנות משנה – תנאי קדם

- נניח כי במחלקה C מופיע הקוד הבא:

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

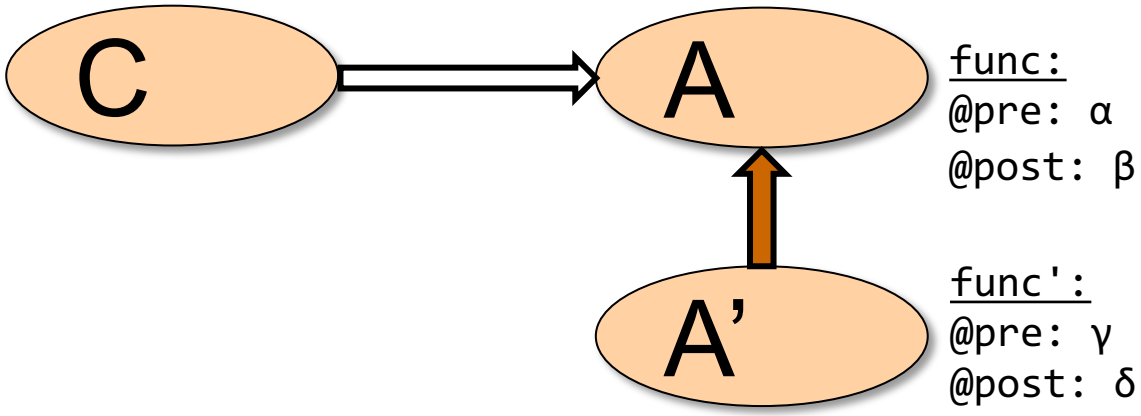
- על C לקיים את תנאי הקדם של A.r(): היא כלל אינה מכירה את המחלקה A' ואינה יודעת על קיום A'.r()
- לכן על תנאי הקדם המוגדר במחלקה הנגזרת להיות **שווה** או **חלש יותר** מתנאי הקדם המקורי





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 ונירואות

- 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]

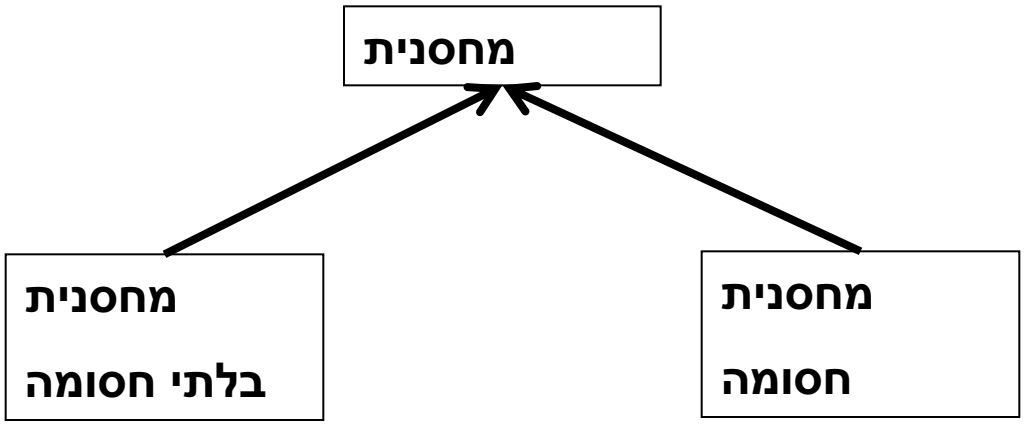
- לדוגמא: בהנתן מימוש המחלקה A, אילו מבין הגירסאות של func ניתן להוסיף ל B שיורשת מ A?

```
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***

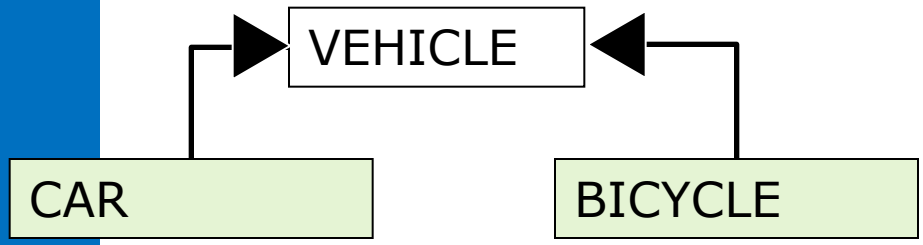
- למרות שתנאי הקדם הקונקרטי אכן מתחזק ע"י המחסנית החסומה תנאי הקדם המופשט נשאר ללא שינוי

תנאי קדם מופשט

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

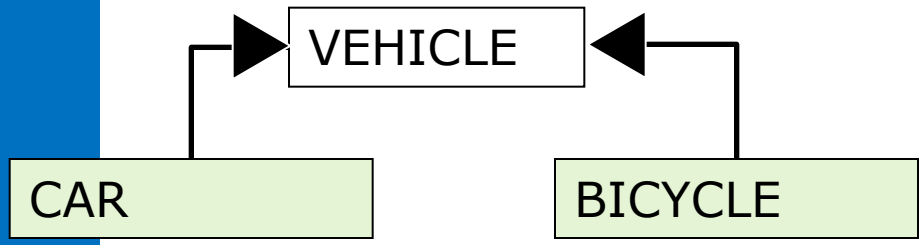
ראייה לטווח רחוק

- האבולוציה של היררכית מחלקות כלי הרכב לא מתחילה בגזירת מחלקות קונקרטיות שיירשו מ VEHICLE
- הגיוני יותר שבמהלך מימוש \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



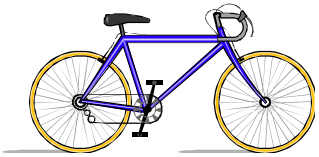
דוגמא

- 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 מיורשותיה?



פתרון

- מתודה בולאנית כגון canGo() תעשה את העבודה
- המתודה תוגדר כמחזירה TRUE עבור VEHICLE (או שתוגדר כ abstract), ועבור כל אחת מיורשותיה 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 is-part-of או has-a
  - 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

מלבן לא יורש מריבוע

is-a לא מתקיים יחס

```
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;
    }
    ...
}
```

- ברור כי העיצוב לקוי – Rectangle is **NOT** a Square
- למשל Square צריך להכיל את **המשתמר** של `getLength()==getWidth()`
  - וברור כי `Rectangle` לא שומר על כך

אז אולי ריבוע יורש ממלבן ?

לא מתקיים עקרון ההחלפה!

- (ריבוע הוא מלבן) אבל is-a מתקיים יחס במימוש הספציפי הזה לא מתקיים **עקרון ההחלפה**.
- לא ניתן להשתמש בריבוע בכל הקשר שבו ניתן היה להשתמש במלבן
- זה מפתיע – מכיוון שמתמטית ריבוע הוא סוג של מלבן
- אז איך בכל זאת נממש את המחלקות ריבוע ומלבן?
- בעולם התוכנה יש לעשות "ויתורים כואבים"

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

    public double getWidth(){
       return width;
    }

    public double getLength(){
       return length;
    }

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

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

    ...
}
```

העמסה והורשה

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 **שבריר**י

העמסה והורשה - שבריריות

```java
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

תבנית העיצוב Bridge

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 {
...
}
```

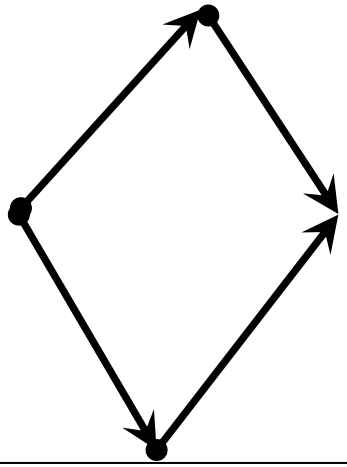
אין ב Java הורשה מרובה

- 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;}
}
```

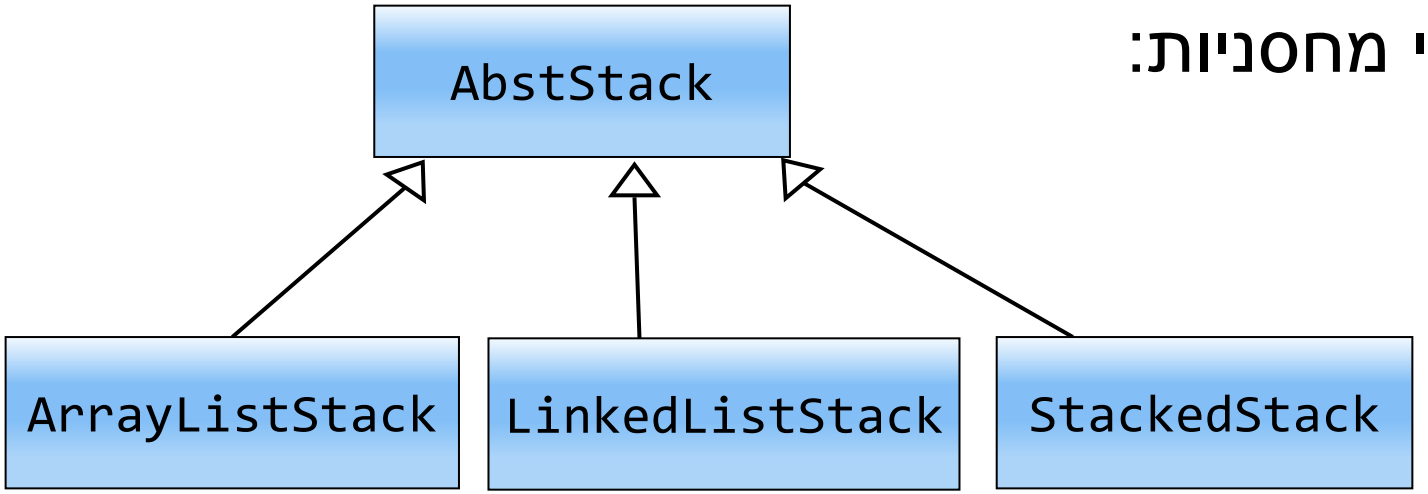
```
interface Driver {
...
}
```



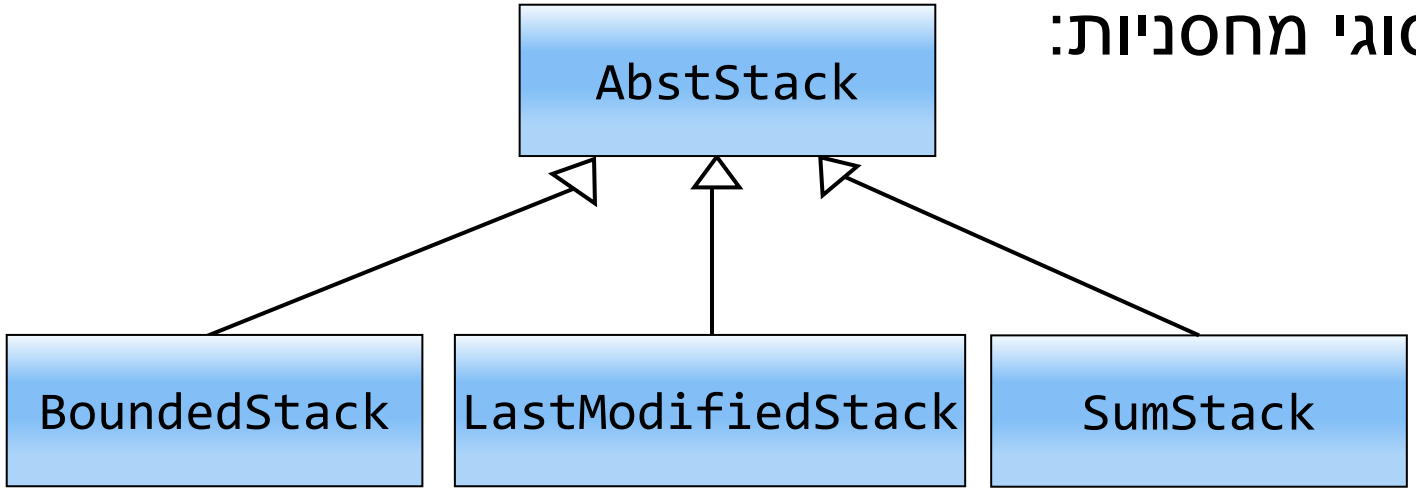
```
Class OpportunisticDriver
extends GoodDriver, BadDriver
(not possible)
```

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

- סוגי מחסניות:

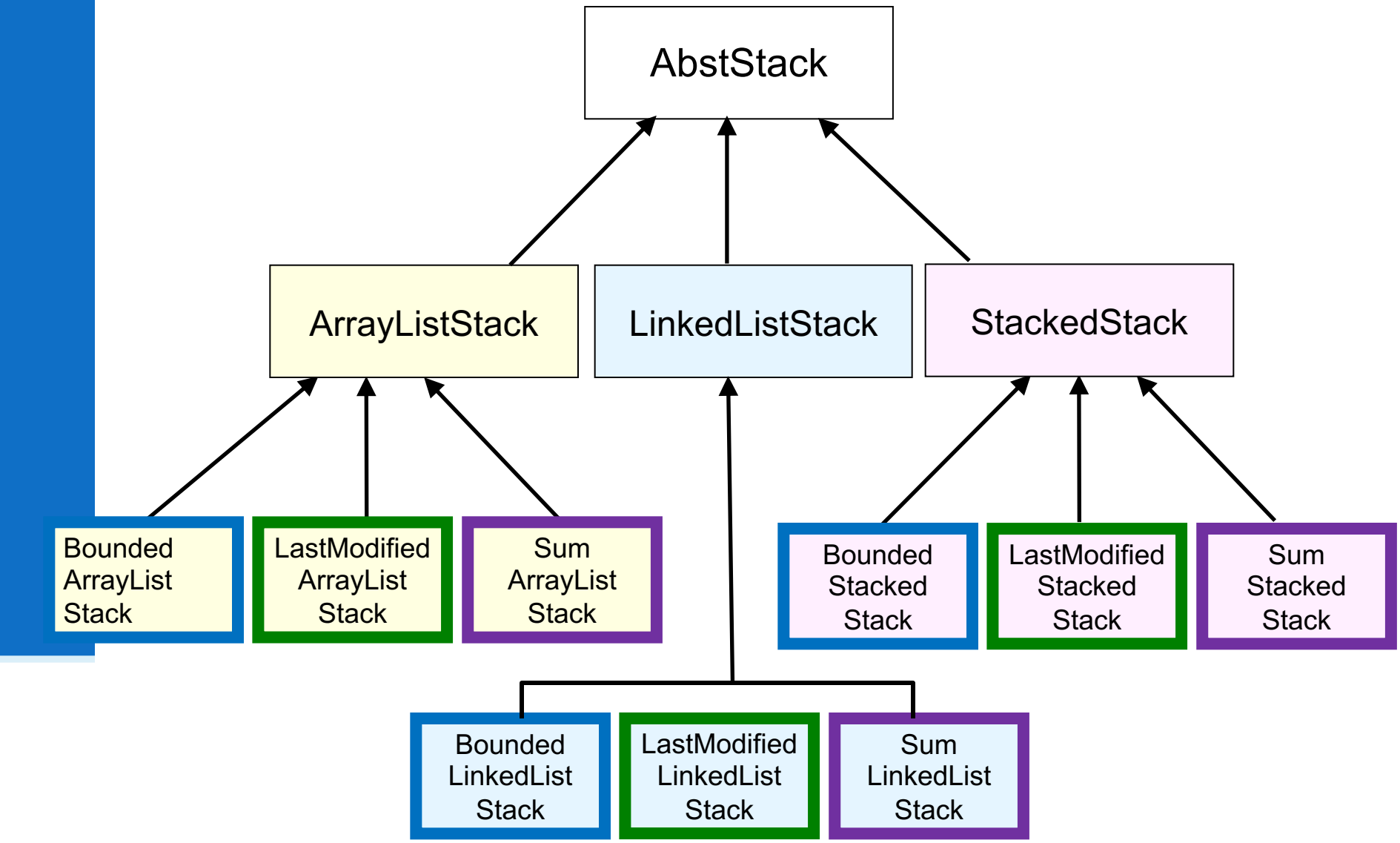


- עוד סוגי מחסניות:



ילדים זה שמחה?

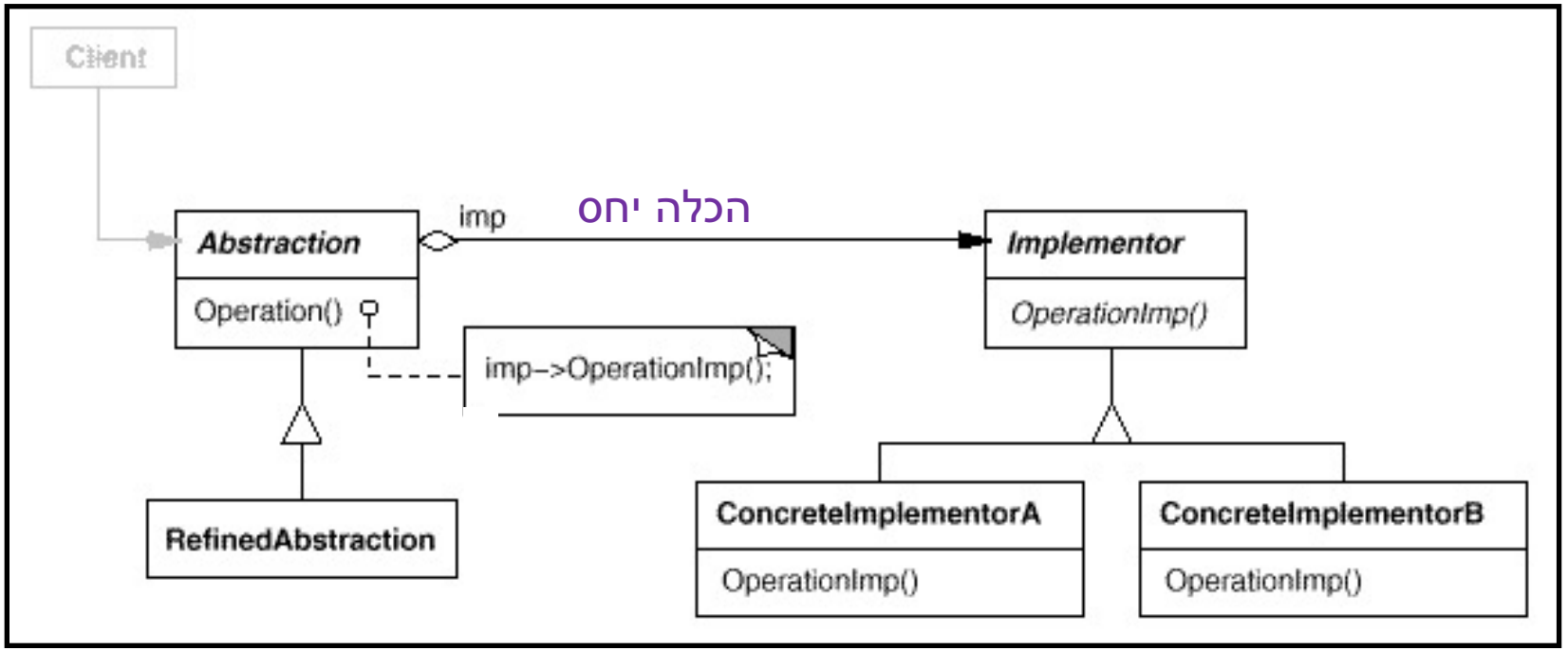
- סוג ההורשה של 3 המחלקות העליונות שונה מסוג ההורשה של 3 המחלקות התחתונות
- מה יקרה אם נרצה למשל: **SumArrayListStack** ?
- בשפות מסוימות (כגון C++ או Eiffel) ניתן ליצור מחלקה חדשה היורשת משתיהן
  - הדבר פותח פתח למכפלה קרטזית (9 מחלקות!) שתבטא את כל הצירופים האפשריים
  - דבר זה ייצור אינפלציה של מחלקות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


AbstStack
ArrayListStack
LinkedListStack
StackedStack
Bounded ArrayList Stack
LastModified ArrayList Stack
Sum ArrayList Stack
Bounded Stacked Stack
LastModified Stacked Stack
Sum Stacked Stack
Bounded LinkedList Stack
LastModified LinkedList Stack
Sum LinkedList Stack

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 ייצוג (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 ייצוג שהוא

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

- נשים לב להבדל שבין המנשק **IStack** ובין המנשק **IStackImpl**
  - המנשק **IStack** מייצג את המחסנית
  - המנשק **IStackImpl** מייצג את **מימוש** או **ייצוג** המחסנית
- המחלקה **SimpleStack** המממשת את **Istack מכילה** מופע של מחלקה המממשת את **IStackImpl**
  - הורשה (מימוש) לצורכי ייצוג תתבצע מ **IStackImpl**
  - הורשה (מימוש) הנוגעת להפשטה תתבצע מ **Istack**

דוגמא למימוש המחסנית באמצעות ArrayList

```
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?

תבנית העיצוב Bridge

- אז מה יש לנו עד כה?
- שני מנשקים שמאפשרים לנו לייצר כל שילוב בין התנהגות לייצוג.
- 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 (Bounded, LastModified)

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

}
```

מתאר ייצוג (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.

בחינה

- החומר לבחינה – כל החומר, כולל תרגולים ותרגילי בית
- מחצית מהבחינה – 2 שאלות פתוחות, ומחצית מהבחינה שאלות אמריקאיות
  - באתר הקורס ניתן למצוא דוגמאות לבחינות עבר עם פתרונות.
- פורום שאלות מבחינות קודמות
- אופן המענה על הבחינה