

תוכנה 1 בשפת Java  
שיעור מספר 9: "ירושה נכונה" (הורשה II)

**ליאור וולף**

בית הספר למדעי המחשב  
אוניברסיטת תל אביב

# ראשית חזרה על מתודות סטאטיות

---

■ רב המתודות שנכתוב לא סטטיות..

```
public class MrHappyObject {
    private String _mood = _HAPPY;
    private final static String _HAPPY    = "happy";
    private final static String _ANNOYED = "annoyed";
    private final static String _ANGRY    = "angry";
    public void printMood() {
        System.out.println( "I am " + _mood );
    }
    public void receivePinch() {
        if( _mood.equals( _HAPPY ) ) {
            _mood = _ANNOYED;
        } else {
            _mood = _ANGRY;
        }
    }
    public void receiveHug() {
        if( _mood.equals( _ANGRY ) ) {
            _mood = _ANNOYED;
        } else {
            _mood = _HAPPY;
        }
    }
}
```

```
MrHappyObject obj1 = new MrHappyObject();
MrHappyObject obj2 = new MrHappyObject();
obj1.printMood();
obj2.printMood();
```

```
obj1.receiveHug();
obj2.receivePinch();
obj1.printMood();
obj2.printMood();
```

```
private static int _instantiations;
public MrHappyObject() {
    _instantiations++;
}
public static int instances() {
    return _instantiations;
}
```

Why Static?

# היום בשיעור

- תבניות עיצוב (Template Method, Builder)
- מידע על טיפוסים בזמן ריצה
- תבניות וירושה
- קבלנות משנה (ירושה והחזקה)
- שימוש לרעה בירושה

# אלגוריתם כללי

## Template Method Design Pattern

- מחלקות מופשטות 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, תבנית
  - המתודות המופשטות מגדירות אבני בניין (hooks) שבעזרתן ניתן יהיה לממש את האלגוריתמים הכלליים במחלקות היורשות
  - שימו לב – הטרמינולוגיה הפוכה!
- דוגמא: מימוש המתודה changeTop במחשנית לא מחייב הכרות עם המחשנית עצמה

# מחסנית מופשטת

```
abstract class AbstStack <T> implements IStack<T> {
```

```
    public void change_top(T t) {  
        pop ();  
        push(t);  
    }  
}
```

```
abstract public void push(T t);  
abstract public void pop();  
}
```

- השרות `change_top` אינו תלוי במימוש של `push` או `pop` אלא רק בחוזה שלהם
- `change_top` מכונה אלגוריתם כללי
- `pop` ו-`push` הם `hooks` או `callbacks`

# 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).
- עוד דוגמאות בשיעורי הבית
- **זוהי תבנית עיצוב** – 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 ניתן לממש בה את האלגוריתם הכללי



# ירושה מרובה

- מנגנון הירושה נועד לתאר בצורה נכונה יחסים בין מחלקות המבטאות ישויות (טיפוסים) בעולם האמיתי
- לפעמים יש הצדקה לירושה 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

# ירושה 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);
}

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

שגיאת קומפילציה  
ב Java אין דבר כזה!

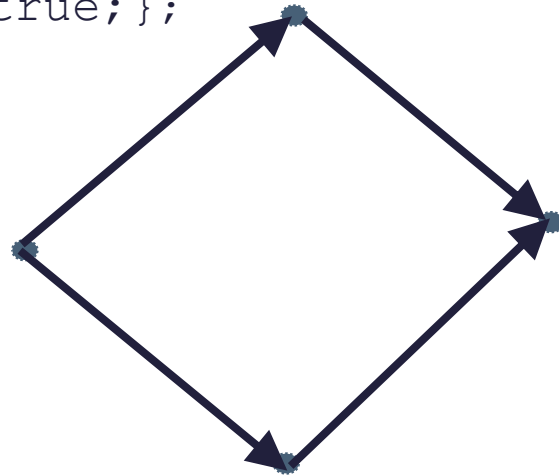
■ ולכן הגיוני אולי שיירש משתיהן:

■ ממי יורשת המחלקה Float?

# מה הבעיה בירושה מרובה?

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

```
interface driver {  
...  
}
```



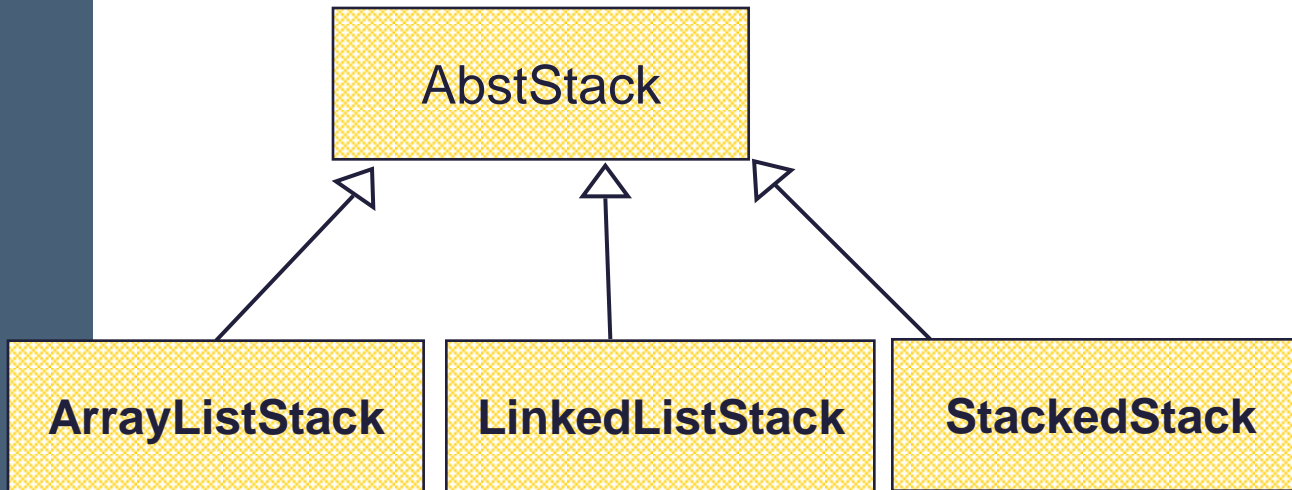
Class OpportunisticDriver  
extends GoodDriver, BadDriver  
(not possible)

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

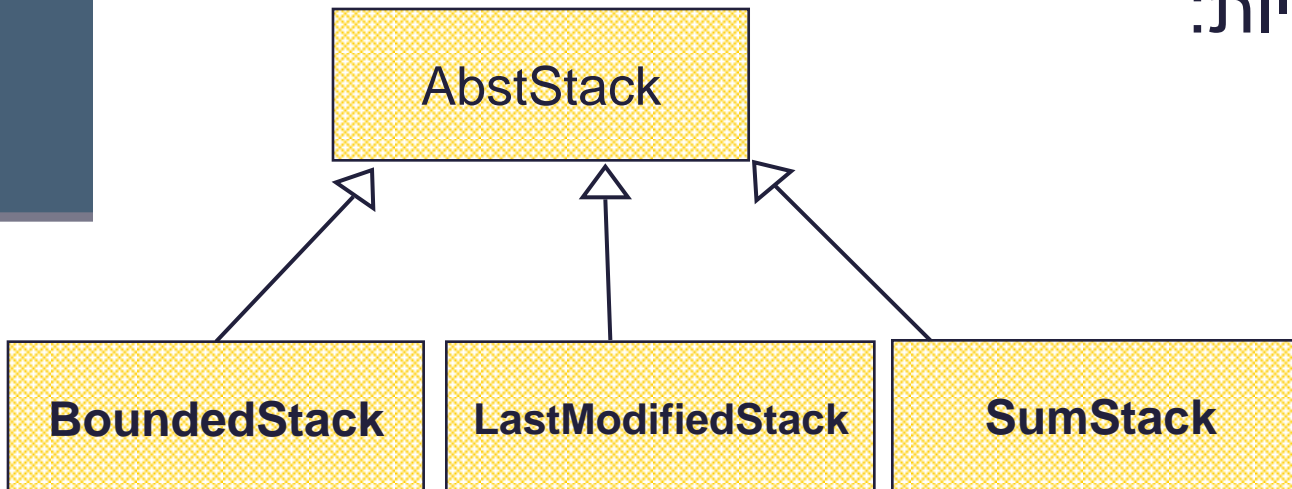
# אין ב Java ירושה מרובה

- אין ב Java ירושה 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 הירושה המרובה
- **Bridge Design Pattern** – פיתוח מערכת מחלקות היררכית, כאשר לאחת המחלקות צאצאים מסוגים שונים

## סוגי מחסניות:

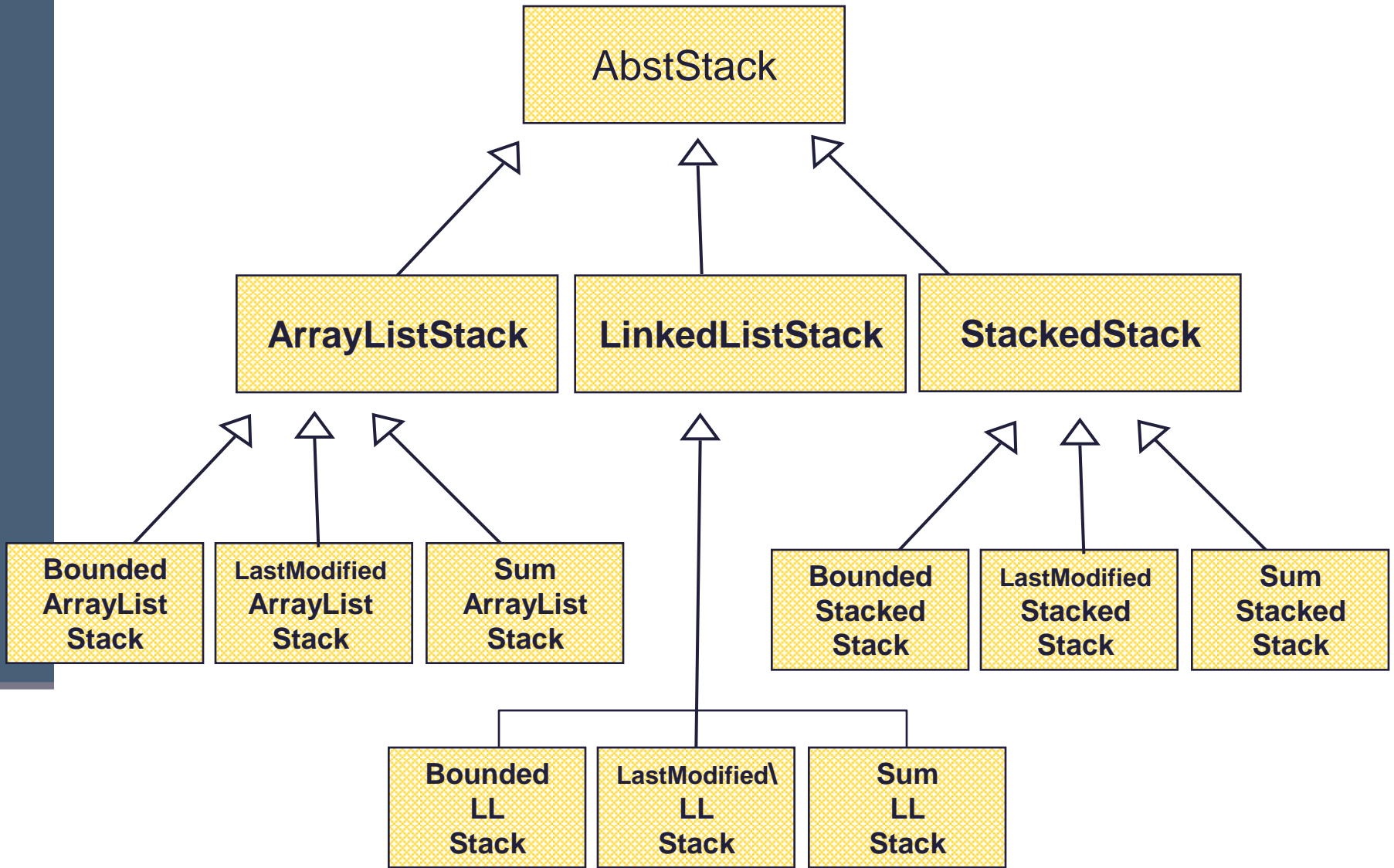


## עוד סוגי מחסניות:



# ילדים זה שמחה

- סוג הירושה של 3 המחלקות העליונות שונה מסוג הירושה 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 ירושה (לדוגמא את `SumArrayListStack`) ב `Java` ?

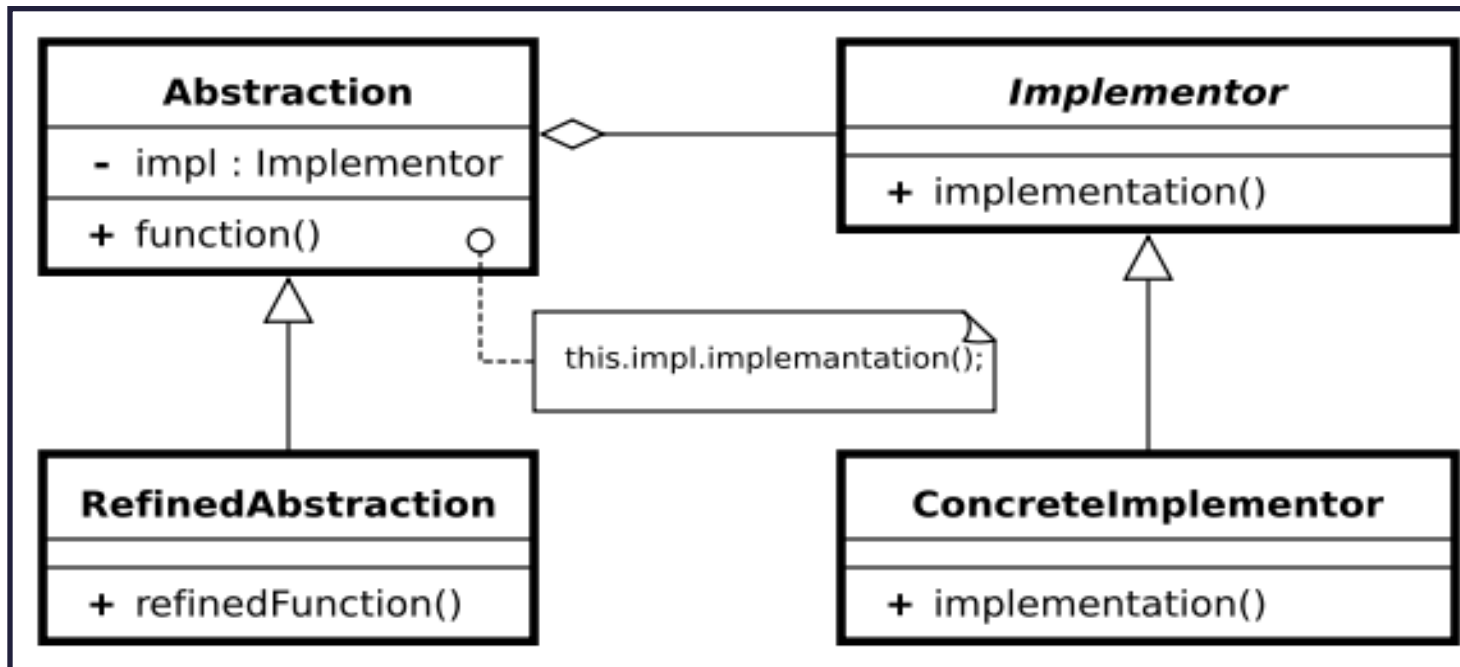


# לא כל כך שמחה

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



# תרשים מחלקות



```
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;
    }
}

```

```
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 המחסנית

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

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

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

## • דוגמא למימוש מחסנית בעזרת 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?

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

Bug  
before we  
assume 0 is  
the latest

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

תוכנה 1 בשפת Java  
אוניברסיטת תל אביב

# טיפוסי זמן ריצה

- בשל הפולימורפיזם ב Java אנו לא יודעים מה הטיפוס המדויק של עצמים
- הטיפוס הדינאמי עשוי להיות שונה מהטיפוס הסטטי
- בהינתן הטיפוס הדינאמי עשויות להיות פעולות נוספות שניתן לבצע על העצם המוצבע (פעולות שלא הוגדרו בטיפוס הסטטי)
- כדי להפעיל פעולות אלו עלינו לבצע המרת טיפוסים (Casting) על ההפניה

# המרת טיפוסים Cast

המרת טיפוסים בג'אווה נעשית בעזרת אופרטור אונרי שנקרא Cast ונוצר על ידי כתיבת סוגריים מסביב לשם הטיפוס אליו רוצים להמיר.

(Type) <Expression>

(הדיון כאן אינו מתייחס לטיפוסים פרימיטיביים).

הוא מייצר ייחוס מטיפוס Type עבור העצם שהביטוי <Expression> מחשב, אם העצם מתאים לטיפוס.

הפעולה מצליחה אם הייחוס שנוצר מתייחס לעצם מתאים לטיפוס Type

המרה למטה (downcast): המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם.

המרה למעלה (upcast): המרה של ייחוס לטיפוס יותר כללי (מחלקה או ממשק)

כל המרה אחרת גוררת שגיאת קומפילציה.

המרה למעלה תמיד מצליחה, ובדרך כלל לא מצריכה אופרטור מפורש; היא פשוט גורמת לקומפיילר לאבד מידע

המרה למטה עלולה להיכשל: אם בזמן ריצה טיפוס העצם המוצבע לא תואם לטיפוס Type התוכנית תעוף (ייזרק חריג ClassCastException)

# טיפוסי זמן ריצה

- תעופת תוכנית היא דבר לא רצוי – לפני כל המרה נרצה לבצע בדיקה, שהטיפוס אכן מתאים להמרה

- יש לשים לב כי ההמרה ב Java אינה מסירה או מוסיפה שדות לעצם המוצבע (בשונה מ slicing בשפת C++ למשל)

- בזמן קומפילציה נבדק כי ההסבה אפשרית (compatible types)

- ואולי מתבצע שינוי בטבלאות השרותים שמחזיק העצם (נושא זה ילמד בשיעור 11)

- כאמור בזמן ריצה המרה לא חוקית תיכשל ותזרוק חריג בדוגמא הבאה השאילתא () `maxSide` מוגדרת רק למצולעים (ומחזירה את אורך הצלע הגדולה ביותר). אין כמובן שאילתא כזאת במחלקה `Shape` (גם לא מופשטת).

- כשהלקוח רוצה לחשב את אורך הצלע הגדולה ביותר מבין כל הצורות במערך, על הלקוח לברר את טיפוס העצם שהועבר לו בפועל ולבצע המרה בהתאם



# טיפוסי זמן ריצה

- דרך אחת לבצע זאת היא ע"י המתודה `getClass` המוגדרת ב-`Object` והשדה הסטטי `class` הקיים בכל מחלקה:

```
Shape [] shapeArr = .....
double maxSide = 0.0;
double tmpSide;
for (Shape shape : shapeArr) {
    if (shape.getClass() == Polygon.class) {
        tmpSide = ((Polygon) shape).maxSide();
        if (tmpSide > maxSide)
            maxSide = tmpSide;
    }
}
```

מה לגבי צורות מטיפוס  
Rectangle או Triangle ?

עצמים אלה אינם מהמחלקה  
Polygon ולכן לא ישתתפו

# instanceof

■ האופרטור `instanceof` בודק האם הפנייה `is-a` מחלקה כלשהי - כלומר האם היא מטיפוס אותה המחלקה או ירשיה או מממשיה

```
Shape [] shapeArr = ....
double maxSide = 0.0;
double tmpSide;
for (Shape shape : shapeArr) {
    if (shape instanceof Polygon) {
        tmpSide = ((Polygon) shape).maxSide();
        if (tmpSide > maxSide)
            maxSide = tmpSide;
    }
}
```

# instanceof

- שימוש ב-Casting בתוכניות מונחות עצמים מעיד בדר"כ על בעיה בתכנון המערכת ("באג ב-design") שנובעת לרוב משימוש לא נכון בפולימורפיזם
- לעיתים אין מנוס משימוש ב-Casting כאשר משתמשים בספריות תוכנה כלליות אשר אין לנו שליטה על כותביהן , או כאשר מידע הלך לאיבוד כאשר נכתב כפלט ואחר כך נקרא כקלט בריצה עתידית של התכנית.
- הדוגמא שניתנה היא נדירה ואולי לא מציאותית.



# תבניות וירוושה

# 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?

■ ניהח 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
- הפרטים מסתבכים בגלל האינטראקציה בין מנגנון ההכללה ובין יחס הירושה (is-a-ה יחס)
- 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));

# 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: אינו סוג של FCStack<String> ■

זה לא אינטואיטיבי אבל נכון. ■

# הכללה ויחס is-a (המשך)

- ההשמה `ts = to` לא חוקית (שגיאת קומפילציה).
- לעומת זאת זה בסדר (רק תחבירית!):

```
String [] as = new String[5];  
Object [] ao = as;
```

- שימוש שגוי במערך יחולל שגיאת זמן ריצה:

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

- השימוש בטיפוסים מוכללים סותם פרצה זו בתחביר המקורי של שפת Java

- לא ניתן ליצור מערך גנרי (בגלל מחיקת הטיפוס T בזמן ריצה):

```
content = new T[capacity] // compile error
```

- אבל זה כן (עם `Type Safety Warning`):

```
content = (T[])new Object[capacity];
```



# 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();
```

```
//same as: Stack<?> raw = new FCStack<Object>();
```

```
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 הממשק 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 יאפשר להשתמש בגוף המחלקה הגנרית בשרותים 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<String>` ובין עצם מטיפוס `FCStack<Integer>`, 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(PrintWriter out, List list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        out.print(list.get(i).toString());  
    }  
}
```

- כדי להימנע מאזהרות קומפילציה נשנה את List לטיפוס מוכלל:

```
public static void printList(PrintWriter out, List<Object> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        out.print(list.get(i).toString());  
    }  
}
```

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



# ג'וקרים

■ נשתמש בג'וקר (סימן שאלה - ?)

```
public static void printList(PrintWriter out, List<?> list) {  
    for(int i=0, n=list.size(); i < n; i++) {  
        if (i > 0) out.print(", ");  
        Object o = list.get(i);  
        out.print(o.toString());  
    }  
}
```

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

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

■ יש גם חסמים תחתונים ושרותים מוכללים:

```
public static Myclass addAll(Collection<? super Myclass> c, Myclass a)
```

# סיכום generics

- מנגנון ההכללה מאפשר להימנע מהמרות בלי לשכפל קוד
- קוד שאין בו המרות מפורשות ושאינן 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](#)

**קבלנות משנה -  
על ירושה, טענות וחוזים**



# ירושה 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

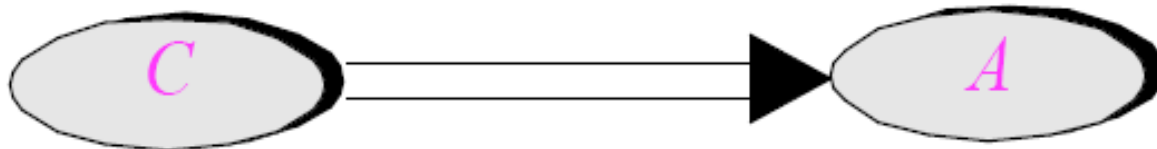
■ מחלקת C היא לקוחה של מחלקה A, כלומר:

■ יש ל-C הפנייה ל-A (אחד השדות)

או

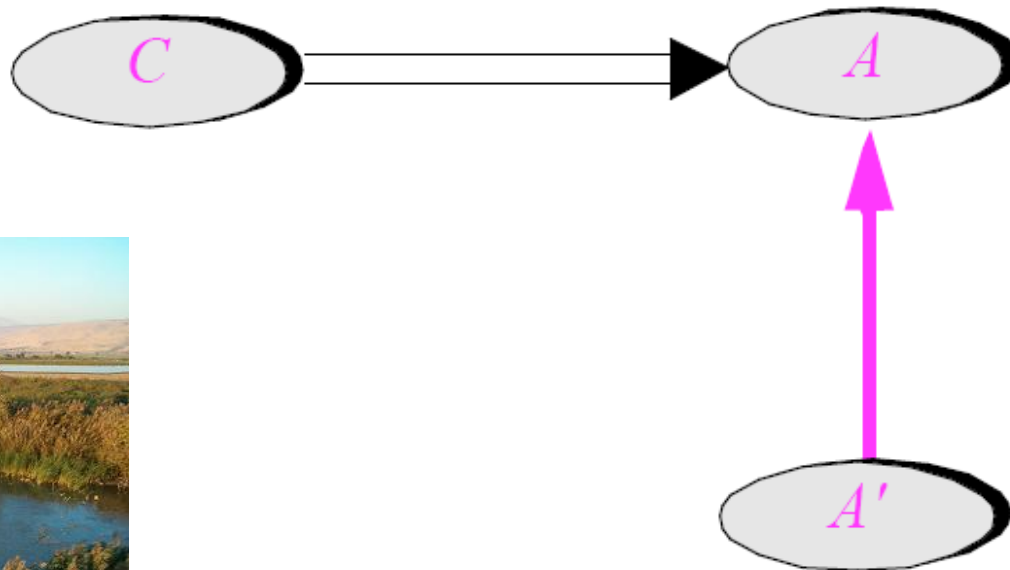
■ אחת המתודות של C מקבלת פרמטר מטיפוס A (הפנייה ל A)

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



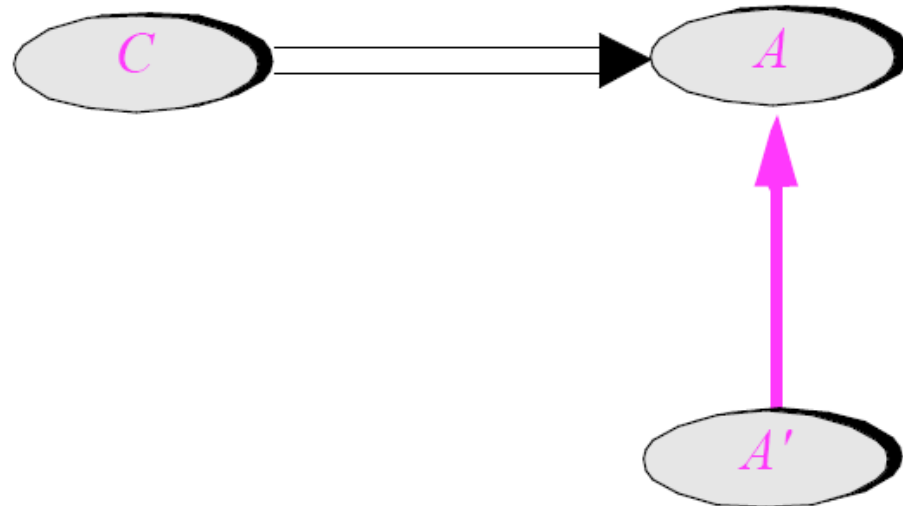
# קבלנות משנה - השמורה

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



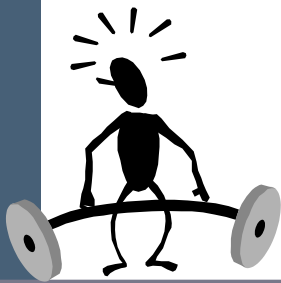
# קבלנות משנה – תנאי קדם ובתר

- המחלקה  $A'$  דורסת (overrides) רוטינה  $r()$  של  $A$
- מה יש לדרוש מתנאי הקדם והבתר של המתודה החדשה ביחס לאלו של הרוטינה המקורית?



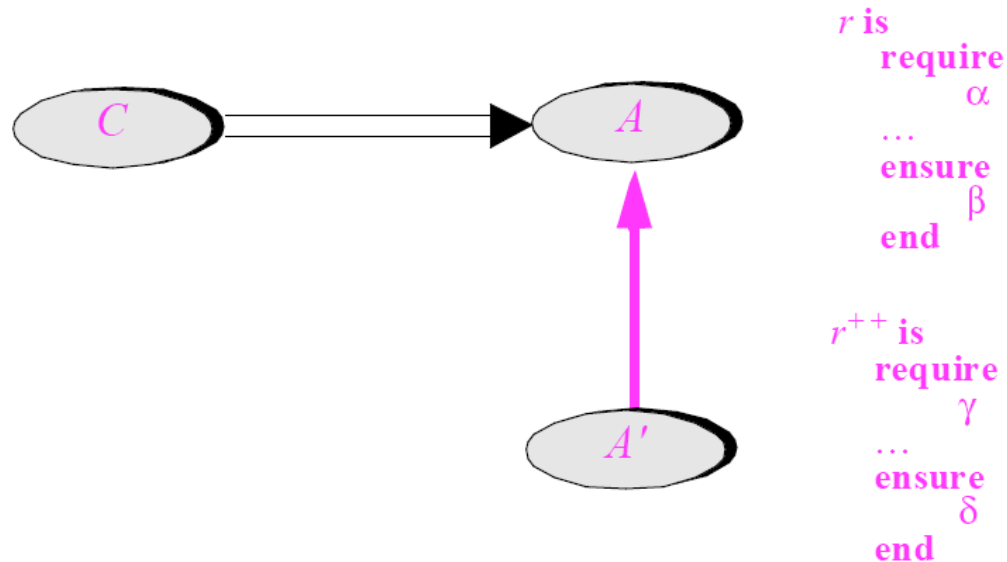
$r$  is  
require  
     $\alpha$   
...  
ensure  
     $\beta$   
end

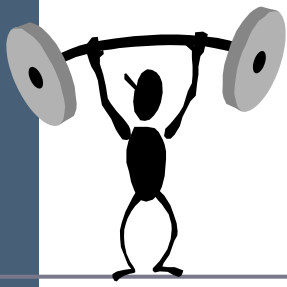
$r^{++}$  is  
require  
     $\gamma$   
...  
ensure  
     $\delta$   
end



# קבלנות משנה – תנאי קדם

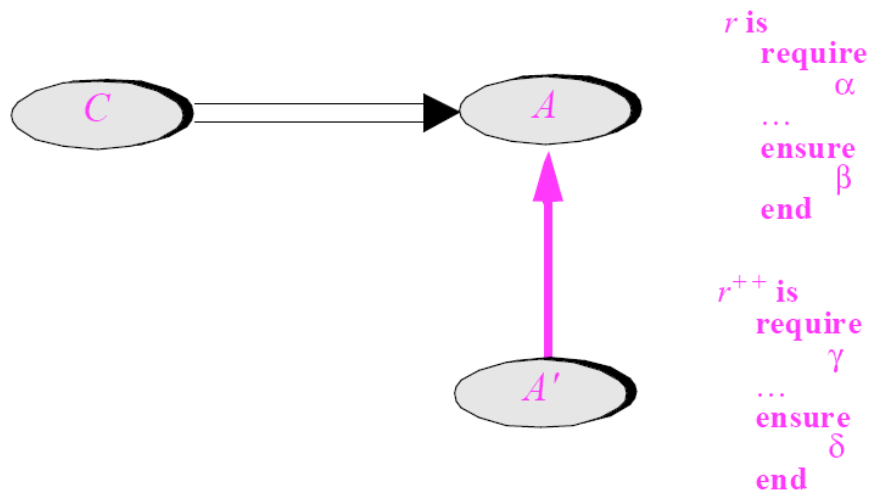
- נתבונן בקריאה  $1.x()$  המופיעה במחלקה C
- על C לקיים את תנאי הקדם של  $A.r()$ , היא כלל אינה מכירה את המחלקה  $A'$  ואינה יודעת על קיום  $A'.r()$
- לכן על תנאי הקדם המוגדר במחלקה הנגזרת להיות שווה או חלש יותר מתנאי הקדם המקורי





# 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)



# השמורה האפקטיבית

- השמורה ה'אמיתית' של מחלקה מורכבת מ AND לוגי של כל הטענות המופיעות בשמורת אותה מחלקה ובכל הוריה לאורך עץ הירושה
- אם עבור רמה (מחלקה) מסוימת בעץ הירושה לא הוגדרה שמורה, ניתן להתייחס לשמורה שלה כ- TRUE
- כותב מחלקה יכול להגדיר את השמורה שלה בצורה מרומזת (implicit) ע"י ציון הטענות החדשות בלבד

# תנאי קדם אפקטיבי

- 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 הירושה
- אם עבור רמה (מחלקה) מסוימת בעץ הירושה לא הוגדר תנאי קדם למתודה זו, ניתן להתייחס לתנאי הקדם שם כ- FALSE
- עקרון זה לא תופס עבור מחלקת הבסיס. מדוע?
- כותב תנאי הקדם של המתודה שהוגדרה מחדש במחלקה כלשהי, יכול להגדיר אותו בצורה מרומזת (implicit) ע"י ציון הטענות החדשות בלבד



# תנאי בתר אפקטיבי

- תנאי הבתר ה'אמיתי' של מתודה שהוגדרה מחדש במחלקה כלשהי הוא ה AND הלוגי של כל תנאי הבתר של מתודה זו בכל הוריה של אותה מחלקה לאורך עץ הירושה
- אם עבור רמה (מחלקה) מסוימת בעץ הירושה לא הוגדר תנאי קדם למתודה זו, ניתן להתייחס לתנאי הקדם שם כ- TRUE
- כותב תנאי הבתר של המתודה שהוגדרה מחדש במחלקה כלשהי יכול להגדיר אותו בצורה מרומזת (implicit) ע"י ציון הטענות החדשות בלבד

# דוגמא

```
public class MATRIX {  
    ...  
    /** inverse of current with precision epsilon  
     * @pre epsilon >= 10 ^(-6)  
     * @post (this.mult($prev(this)) - ONE).norm <= epsilon  
     */  
    void invert(double epsilon);  
    ...  
}
```



# דוגמא



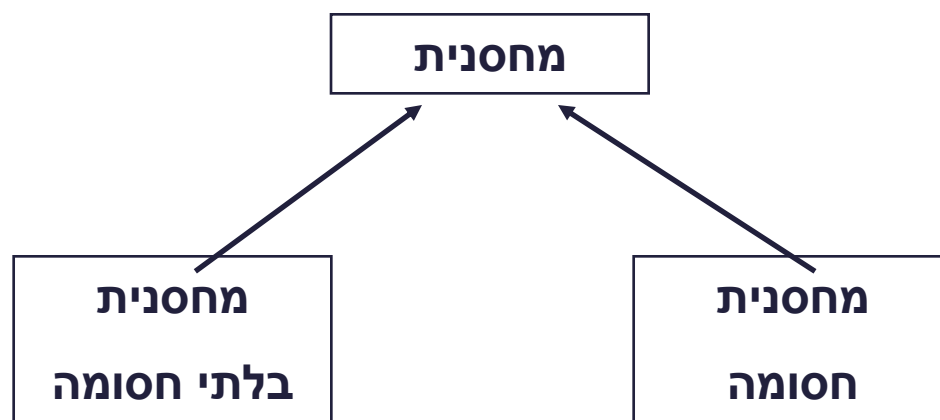
```
public class ACCURATE_MATRIX extends MATRIX {  
    ...  
    /** inverse of current with precision epsilon  
     * @pre epsilon >= 10(-20)  
     * @post (this.mult($prev(this)) - ONE).norm <= epsilon/2  
     */  
    void invert(double epsilon);  
    ...  
}
```

# עוד על ירושה וחוזים

- בנוסף לחריגים, שלגביהם ג'אווה מקפידה על כללי החוזה בירושה, יש עוד כללים בשפה שנובעים משיקולי חוזה וירושה:
- למתודה הדורסת [הממשת] **מותר להקל** את הנראות – כלומר להגדיר סטטוס נראות רחב יותר, אבל אסור להגדיר סטטוס נראות מצומצם יותר.
- (מגירסא 5) למתודה הדורסת [הממשת] **מותר לצמצם** את טיפוס הערך המוחזר, כלומר טיפוס הערך המוחזר הוא תת טיפוס של טיפוס הערך המוחזר במתודה במחלקת הבסיס שלה [במנשק]

# תנאי קדם מופשט

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



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

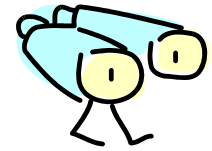
# תנאי קדם מופשט

- תנאי הקדם לא יכול להיות ריק (TRUE) כי אז הוא יחזק ע"י המחסנית החסומה
- תנאי הקדם צריך להיות `!full()` כאשר `full()` היא מתודה מופשטת (או מתודה המחזירה תמיד `false`) שתוגדר מחדש במחלקה מחסנית חסומה להחזיר `count() == capacity()`
- תנאי קדם המכיל מתודות מופשטות או מתודות שנדרסות במורד הירושה נקרא **תנאי קדם מופשט**
- למרות שתנאי הקדם הקונקרטי אכן מתחזק ע"י המחסנית החסומה תנאי הקדם המופשט נשאר ללא שינוי

# תנאי קדם מופשט

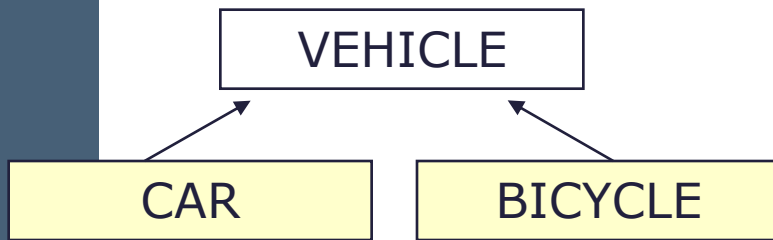
- כאשר מחלקת הבסיס מופשטת, תנאי קדם טריוויאליים מחייבים לפעמים **ראייה לעתיד**, כדי שלא יחזקו במחלקות נגזרת
- ראייה לעתיד אינה דבר מופרך במחלקות מופשטות
- נתבונן בדוגמא נוספת: מערכת תוכנה אשר מיוצגים בה כלי תחבורה שונים כגון מכונית, אווירון ואופניים

# 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

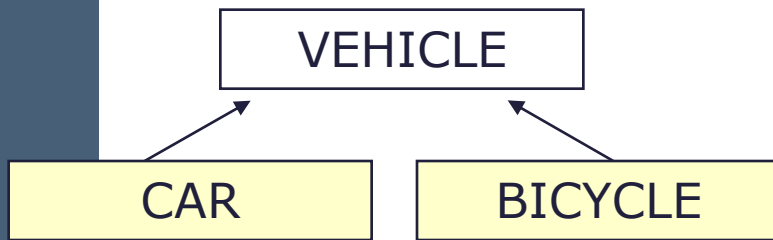




- מהו תנאי הקדם של המתודה `go()` של המחלקה `VEHICLE` ?
- על פניו – אין כל תנאי קדם לפעולה מופשטת
- מה עם המחלקה `CAR` ? – לה בטח יש דרישות כגון `hasFuel()`
- מה עם המחלקה `BICYCLE` ? – לה בטח יש דרישות כגון `hasAir()`
- איך `VEHICLE` תגדיר תנאי קדם ל `go()` גם כללי מספיק וגם שלא יחוזק ע"י אף אחד מירשותיה?



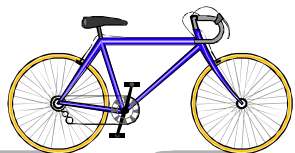
# פתרון



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

- המתודה תוגדר כמחזירה `TRUE` עבור `VEHICLE` (או שתוגדר כ `abstract`), ועבור כל אחת מירשותיה תוגדר לפי המחלקה האמורה

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



# לפעמים ירושה זה רע

- ירושה היא מנגנון אשר חוסך קוד ספק
- פרט למנגנון הרב-צורתיות (polymorphism) ירושה היא סוכר תחבירי של הכלה ואינה הכרחית
  - במקום ש B יירש מ-A , ל-B יכולה להיות התכונה A (שדה)
- יחסי ירושה 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
- ירושה או מופע?
  - האם Massachusetts יורשת מ-State?

# הכוח משחית

■ על המחלקה היורשת לקיים את 2 העקרונות:

■ is-a on יחס

■ עקרון ההחלפה

■ אי שמירה על כך תגרום לעיוותים במערכת התוכנה

■ לדוגמא: ננסה לבטא את יחס המחלקות Rectangle

- Square בעזרת ירושה

Not is-a Relation

# 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;  
    }  
    ...  
}
```

ברור כי העיצוב לקוי – Rectangle is NOT a Square ■

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

Substitution principle doesn't hold!

# אז אולי ריבוע יורש ממלבן?

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

■ מתקיים יחס is-a אבל 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?

```
    public void widen(double delta) {  
        width += delta;  
    }
```

■ בעולם התוכנה יש לעשות "ויתורים כואבים"

```
        ...  
    }
```