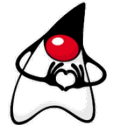


# תכנות מונחה עצמים - סיכום הרצאות

22 ביוני 2025



\* הסיכום נכתב תוך כדי השיעור וייתכן שנפלו בו טעויות, על אחריותכם בלבד.  
\* שימו לב שיש בעיה עם האנגלית ולפעמים נכתב הפוך - קראו הפוך  
© גיא יער-און

## הרצאה 1:

### דברים כלליים:

- אם יש לי קובץ הרצה במחשב שלי, ואנסה להעביר אותו למחשב אחר. האם בהכרח ירוץ? לא, ואפילו כנראה שלא. זה תלוי בהרבה דברים ובעיקר באותה אריקטורה ואם יש לי את אותה משפחה של מערכות הפעלה.

- תמיד ב *java* המופעים של מחלקות - למשל מופע של המחלקה *point*, יופיע ב *heap*.

- *this* : נעשה כשנרצה לפנות לאחד השדות מתוך המחלקה, זו לא חובה אך הרבה יותר קריא ומאפשר מצבים של *this.xPoint = xPoint*. בשדות המחלקה כמובן שלא נכתוב *this*.

- ב *java* אין מצביעים! יש *references* - למשל *Point p = newPoint()* המילה השמורה *new* היא רפרנס. מאחורי הקלעים יש עבודת *malloc*. ה *new* יחזיק את הכתובת של האובייקט החדש שיצרנו.

- בקורס שלנו, *class* לרוב יהיו *public*.

- אסור בתכלית האיסור לגשת לשדות מחוץ ל *class*, זה יגרור שגיאת קומפילציה. מה כן? *getters* ו *setters*.

- השדות והמתודות יחד יקראו *classmembers*.

\* נניח ועשינו פעולה *Point p = newPoint(5, 6)* וכעת נרצה לעשות *p = newPoint(2, 3)* זה יהיה חוקי! אין שום שגיאה, ה *garbagecollector* יטפל בזה ו"ינק" את הערכים הקודמים - זה רק יעלה לנו ביעילות.

### מחלקת *String*:

\* פעולה *length()*: מחזירה את האורך של מחרוזת.

**\*פעולה** `indexOf(char)`: מחזיר את האינדקס משמאל של `char`, אם לא מוצא יחזיר -1.

- יש לשים לב שב `java` המחלקה היא אובייקט, ובדומה גם מערכים הם אובייקט (נעיר בהמשך)

- שני פעולות מיוחדות ששווה להתעכב עליהם הם `s.toString` ו `s.equals`. הן שתי פעולות שקיימות בכל מחלקה ויש לדרוס אותן ולממש בעצמנו כיוון שהערך הדיפולטי שלהם לא שווה לכולם.

**הפעולה** `s.equals` בודקת האם הרפרנסים זהים ולא התוכן - מבלבל מאוד ויש לשים לב לזה. למשל עבור הדוגמה הבאה:

```
String t = new string("helloWorld")
```

```
String s = new string("helloWorld")
```

**נקבל** שיודפס סה"כ `false` שכן אמנם התוכן זהה אך הרפרנסים שונים! זה לא מה שציפינו לקבל ולכן נדרוס.

**הפעולה** `toString` מדביקה "האש קוד" מזהה לכל אובייקט ואותו מדפיסה ושוב לא בהכרח יודפס תיאור כפי שחשבנו - גם אותה נדרוס.

## **תכונת** `Encapsulation`:

- בואו נסתכל על דוגמה שגויה:

```
square (Point min, int edge)
{
    this.min = min;
    this.edge = edge;
}
```

קונסטרוקטור זה לא דבר כזה פשוט לפעמים, כאן זה שגוי לגמרי. נשים לב שזה אכן יתקמפל אך יש בעיה בעקרונות של `oop`. נוכל לעשות דרך ה `main.setX` וזה ישנה כאן את ערך הנקודה. מה הפתרון?

לשנות את השורה השניה ל -

```
this.min = new Point(min.getX(), min.getY());
```

וכעת זה מעולה לנו.

למה שראינו כאן קוראים `encapsulation`. נראה כי קיימים מצבים רבים בהם נרצה להגן על השדות שלנו כי ה `private` לא תמיד יעזור, ממצבים כמו כאן חובה להמנע. בהתאם לכך נגדיר גטרים וסטרים גם כמו בדוגמה כאן וזה יותר מלגיטימי.

- יש לשים לב שמחלקת `random` כאשר נשתמש בה בתוכנית, יש לשים אותה כחלק מהשדות.

- בתוך `class` שונים, ניתן לקרוא למתודות בשם זהה.

## **הרצאה 2:**

### **תכונת** `Static – members`:

- סוג מיוחד של `class members`.

**מוטביציה:** לספור כמה פעמים לאורך הפרויקט יצרנו *point*. אם ניצור משתנה רגיל שיהיה חלק מהשדות של המחלקה, בכל פעם הוא יתאפס מחדש כשנכנס למחלקה. לעומת זאת אם נכתוב את השורה הבאה -

```
private static int counter = 0;
```

**כשהדגש** הוא על ה-*static*, כן נצליח לספור את מספר המופעים. זו תכונה ששייכת למחלקה ולא לאובייקט!

**השדה** סטטי הוא שייך לקלאס, ואתחול יתבצע פעם אחת בלבד עם הרצת התוכנית. כלומר, גם אם עוד לא יצרנו אף אובייקט עדיין ונרצה לגשת ולהדפיס את הערך של *counter* נוכל ויצא לנו אפס. למרות שעוד לא יצרנו מופע אחד אפילו של המחלקה.

**כשנרצה** לגשת למתודה סטטית, נעשה זאת דרך השם של המחלקה.

\* ניתן לגשת למשתנים סטטים/ מתודות סטטיות גם דרך אובייקט של המחלקה. אך זה לא נכון! זה יתקמפל אך לא יפה (ולא יפה - לא עושים).

\* מתודה סטטית - דוגמה לא טובה: אם נעשה

```
public static int sum(){  
return x+y;  
}
```

זה כמובן לא יתקמפל. מי אלו בכלל איקס וואי? אין גישה אל משתנים שאלו הם שמותיהם.

\***ספריית math:** הכל בספרייה הוא *static*. נשאלת השאלה - מדוע יש במחלקה מתודות שכלל לא משתמשות בתכונות שלה? התשובה היא שב-*java* חובה להגדיר כל מתודה בתוך מחלקה כלשהי, וזו המחלקה ההגיונית ביותר להכניס אליה את המתודות הנ"ל.

\***דוגמה טובה:** אנו נמצאים במחלקה *A*. ניתן לממש פונקציה כזו במחלקה -

```
A a = new A();  
|
```

**כאשר** *g* היא מתודה במחלקה. כלומר - חוקי לעשות *new* לאותה מחלקה מתוך אותה מחלקה. באשר לדוגמה - היא חוקית כיוון שניתן ליצור כמובן אובייקט מתוך מחלקה, כעת נזמן את "this" שיהיה *a* ואת *g* אין בעיה להפעיל על אובייקט.

\* מתוך מתודה סטטית, לא ניתן לגשת אל מתודה לא סטטית. כיוון שמתודה סטטית שייכת למחלקה בזמן שמתודה שאינה סטטית היא ברמת האובייקט.

\* אם נשתמש הרבה ב-*static* כנראה שלא הבנו נכון את עקרונות *oop* וניתן ללכת חזרה אל המצביעים והכיף ב-*c*. צריך להצדיק שימוש במתודה ומשתנים סטטים! כנראה שבחרנו שפה שאינה נכונה לפרויקט. הצדקה היא מקרים כמו הקאונטר מעלה.

## ממשק - *Interface*: "מי רוצה להתממשק?"

*Polymorphism*: "רב צורתיות" - משה הוא סטודנט. הוא גם אדם, הוא בן זוג ומילואימניק. בעזרת אילו מתודות נתממשק עם משה? משה יתממשק איתנו כנראה כסטודנט. עם המפקד שלו, יתממשק משה כמילואימניק. לאובייקט כזה נקרא פוליפורמיזם.

כעת, נניח ויש לנו מחלקות ריבוע ומעגל. נרצה לממש מתודה שתקבל ריבוע ומעגל. היכן נשים אותה? <== מחלקה חדשה בשם *functions*! חוקי אבל בשביל מה? יש בג'אווה אפשרות חדשה ומעניינת -

```
public interface shape {
    double area();
```

ולכל המחלקות שישתמשו בממשק נוסף `Public class _name implements shape`.

מה קורה כאן? הממשק יעזור לנו והוא יהיה מעין "הצהרת כוונות" כל מי שיממש אותו יהיה חייב לממש את כל הפונקציות שהוא מצהיר עליהן בתוכו, כלומר מחלקת משולש תהיה חייבת להחזיק מתודה שמחשבת שטח משולש ובדומה עבור ריבוע וכל מי שיממש את `shape`. כעת, `shape` הוא דטה-טייפ חדש. רעיון דומה ל-`typedef` בשפת `c`.

כעת נתבונן בדוגמה חשובה -

```
public interface IA {
    double f();
    public class A implements IA{
        ...there is function f and g here
```

ונתבונן בדוגמאות הבאות:

א. שגיאת קומפילציה - `IA a = new IA();`

על מה הוצהר כאן "new"? אין שום בנאי. ניסו ליצור עצם חדש שמשתמש בבנאי שלא קיים - שגיאה כמובן.  
ב. תקין -

```
IA a = new A();
a.f();
```

משתמשים כאן בבנאי שקיים, ויוצרים טיפוס עליו נפעיל פונקציית `f`.  
ג. שגיאה -

```
IA a = new A();
a.f();
a.g();
```

מדוע זו שגיאה? העצם `a` הוא בעצם מסוג הממשק - בממשק אין מתודה `g` שכן קיימת אמנם `A` אך לא בממשק ולכן שגיאת קומפילציה.  
ד.

```
IA a = new A();
a.f();
((A)a).g();
```

חוקי ורץ - יש כאן קאסטינג והפיכת הממשק לעצם של `A`, שם אכן יש מתודה `g`. בעצם מאחורי הקלעים היה כאן בקשה מהקומפילטר - תן לי לעשות `casting` והוא אישר כאן כי זה הגיוני אך באותה המידה יכל שלא לאשר! קאסטינג כאן הוא מסוכן מהסיבה הבאה - אם במקום `new A` היה כאן `new B` היינו מקבלים שגיאת זמן ריצה! מדוע? נניח `A` הוא ריבוע ו-`B` הוא עיגול. היינו ממירים ריבוע לעיגול?! כמובן שלא.

## הערות מתרגול 2 (צביקה ברגר)

\* המחלקה `Integer`: החל מ-`java9` נכתוב כך - `Integer B = 7`.  
נשים לב כי הפעולה הבאה חוקית:

```
int a = 7
Integer b=a
```

\* יש במחלקה תכונות סטטיות `Min - Value` ו-`Max - Value` שמחזיקים ערך מקסימלי/מינימלי שהמחלקה יכולה לקבל.

\* `int` פרמטיבי ולכן הדוגמה הבאה לא חוקית ולא תתקמפל -

```
integer b = null
```

```
int a = b
```

המלצת צביקה - להשתמש במחלקה הזו ולא ב*int*.

\*תכונה מקורס 'תכנות מתקדם' שכן הזכרה בתרגול: נתבונן בדוגמה הבאה -

```
string str1 = "heyWorld"
string str2 = "heyWorld"
str1==str2 ==>> return true!
```

באופן מפתיע יוחזר כאן אמת, תכונה זו נקראת *StringPool*, ג'אווה מחליטה עבור מחרוזות זהות בתוכן שלהן לשמור אותן באותה הכתובת (שימושי כאשר יש שתי מחרוזות ענקיות זהות) - ולכן כיוון שהכתובת כאן זהה יוחזר אמת.

\*יש מלא פעולות של סטרינג - *str.startsWith*, *str.endsWith* וכו'...

\* ניתן לקרוא לקונסטרקטור אחד מקונסטרקטור אחר - נתבונן בדוגמה:

```
stack(){
return this(Max_size);
}
```

מה יש כאן ומדוע זה חוקי ומתקמפל? יש לי מחסנית עם שני קונסטרקטורים: אחד שמקבל גודל ערך מחסנית מקסימלי והשני ריק. את הריק נוכל להגדיר באמצעות זה שמקבל - בעצם המחשב מבין מה עשינו וכיוון שהשתמשנו במילה השמורה *this*, אז הוא הולך לקונסטרקטור שכן קיבל ערך וקורה מה שקורה.

\*הערה חשובה: עבור מערכים אם נעשה *arr1 = arr2* זה יעתיק *byReference* ולכן לרוב נשתמש בלולאת פור על מנת להעתיק מערכים. - בשביל לשמור על *encapsulation*.

### הרצאה 3:

### ירושה - Inheritance

\* נשים לב שלפי עקרונות ה*oop* אם יש שתי מחלקות עם פעולות זהות - לא טוב. מה נעשה? ניצור מחלקה חדשה עם המתודות המשותפות ואז המחלקות האחרות ירשו ממנה. לצורך הדוגמה נרוץ עם המחלקה המשותפת *Employee* ולאלו שירשו ממנה נוסיף ביצירה *extends*. אנחנו נירש כל מה שיש ב*baseClass* ונוכל להוסיף כמובן עוד..

\* **התכונה *override***: אנחנו בתוך מחלקה שירשה ממחלקה אחרת מתודה בשם *f*. כעת, נרצה לדרוס אותה ולממש אותה אחרת. פשוט ואפשרי - נעשה זאת כך:

```
@override
public int f(){
//..}
```

מכאן המסקנה היא שניתן לרשת רק חלק מהמתודות ולא את כולן. נשים לב כי אם דרסנו את רוב המחלקה שירשנו ממנה, מדוע לרשת ממנה מלכתכילה? זה לא נכון!

**תכונת *upCasting***: המרה של אובייקט ממחלקה בת למחלקת אב. למשל - יש לי מחלקת "חיה" ואת מחלקת "כלב" שירשת ממנה. זה "לשדרג" את כלב לתפקיד חיה. (כמובן שזה מאוד מאוד נאמר לצורך הפשטת הרעיון)  
**תכונת *downCasting***: המרה של אובייקט ממחלקת אב למחלקת בת. בואו נראה דוגמה מבלבלת -

```
1. Animal animal = new Dog();
Dog dog = (Dog) animal;
2. Animal animal = new Animal();
Dog dog = (Dog) animal;
```

דוגמה 1 עובדת, מדוע? אמנם החיה היא מטיפוס חיה אך היא מצביעה על כלב ואז בשורה השנייה בדאון קאסט פשוט נגיד לקומפיילר - "אתה רואה אותו עכשיו כחיה, אבל פשוט תתיחס אליו ככלב"

דוגמה 2 לא עובדת ונקבל שגיאה בזמן ריצה - מדוע? גם המשתנה חיה הוא מסוג חיה אך הוא מצביע גם על טיפוס של חיה. איפה יש כלב?! אין כלב כמו קודם. אנחנו מנסים להכריח את הקומפיילר עם דאוןקאסט לומר לו - אתה כלב, והקומפיילר חכם (אבל לא עד כדי כך..). הוא יקנה את זה בהתחלה בקומפילציה, אך בזמן ריצה יכשל וסה"כ נקבל שגיאת זמן ריצה.

## נתבונן כעת בדוגמאות הבאות:

# SANITY TEST



```
public interface IEmployee {
    void work();
    void getSalary(int salary);
}
```

```
public interface Programmer {
    Code program();
}
```

```
public class Employee implements IEmployee {
    public void work() { ... }
    public void getSalary(int salary) { ... }
}
```

```
public class DBA extends Employee { ... }
```

```
public class FrontendProgrammer
    extends Employee implements Programmer { ... }
```

```
public class BackendProgrammer
    extends Employee implements Programmer { ... }
```

## אילו קטעי קוד הינם תקינים ?

1 Employee taylor = new BackendProgrammer();  
taylor.program();

2 Employee taylor = new BackendProgrammer();  
((BackendProgrammer) taylor).program();

3 Employee taylor = new BackendProgrammer();  
((Programmer) taylor).program();

4 Employee noam = new Employee();  
DBA eden = new DBA();  
((Programmer) noam).program();  
((Programmer) eden).program();

5 Employee taylor = new BackendProgrammer();  
((String) taylor).charAt(2);

- נשים לב כי אינו תקין. טיילור הוא *employee* ואין לה מתודה בשם *program*.
- תקין, טיילור אכן בקאנדפרוגרמר ועשינו דאון-קאסט.
- לא תקין, אין קשר אבא-בן בין פרוגרמר ל-*employee*, עם זאת לפי היררכיה *employee* יכול להצביע על אובייקט שמממש פרוגרמר ולכן קומפיילר יאשר קאסטינג. קומפיילר לקח סיכון - וקיבלנו שגיאת זמן ריצה.
- באופן דומה ל-3: קומפיילר לקח סיכון, גילה שאין מתודה כזו ושגיאת זמן ריצה.
- שטות גמורה.

## הורשה משולשת/ממשק משולש

- \* ג'אוה לא מאשרת הורשה משולשת -
- \* ג'אוה כן מאשרת זאת בממשקים -
- בעיית היהלום: נתבונן במצב הבא -

```
public class c extends a,b
```

```
public interface c extends a,b
```

```
public interface A(){
void f();}
public interface B(){
void f();}
```

אם נרצה להוריד משני הממשקים נקבל שגיאה! זה לא חוקי להוריש משני ממשקים עם אותם החתימות (ומשני קלאס בכלל אי אפשר להרחיב..)

## Abstract class

יש מחלקות שלא נרצה שנוכל ליצור טיפוסים חדשים מהם. למשל *employee* היא כזו כי אין מישהו שהוא רק עובד, נרצה ליצור פרודקט מאנג'ר, פרוגרמר וכו'... בשביל למנוע מהמשתמש ליצור אובייקטים מהמחלקה נוסיף לשמה *abstract*. ואז - אם משתמש ירצה להשתמש יקבל שגיאת קומפילציה. כלומר, ניתן למחלקה מקום בפרויקט אך לא ליצור ממנה. במחלקה אבסטרקטית - ניתן ליצור מתודה אבסטרקטית למשל *w()*. עבור כל מתודה -

או שנממש אותה, או שנשים לה *abstract*. קלאס שמרחיב קלאס אבסטרקטי לא חייב להיות אבסטרקטי כמובן. אך מה כן? הוא חייב לממש פעולות אבסטרקטיות של המחלקה - למשל *w()*. \*אם נעשה קלאס וכל המתודות בו אבסטרקטיות זו התנהגות לא נכונה (ואפס במבחן) ==< היה צריך לעשות *Interface*. לסיכום מתודה אבסטרקטית - מתודה בלי גוף - כלומר, רק החתימה שלה מוגדרת, אבל הקוד עצמו לא קיים. נממש את הפעולה עצמה במחלקה יורשת.

## הפעולה super

אנחנו נמצאים במחלקה משולש שווה שוקיים. היא יורשת ממחלקה משולש. בקונסטרקטור - נרצה לבדוק האם אכן המשולש הוא מש"ש. בעייתי לבדוק בקונסטרקטור כי כבר יצרנו אובייקט לא? אז נשלח לקונסטרקטור נקודה עליונה ואורך צלע והוא כבר יצור משולש שווה שוקיים כזה. אך איך בכלל נוכל לתת לו נקודה? אי אפשר כי *private*. איך כן? נשתמש ב - *super*!

```
super(edge.getStart(),edge.getEnd())
```

ופתרנו את הבעיה. כלומר, סופר היא מילה שמורה המאפשרת לגשת לתכונות מחלקת האב מתוך מחלקת הבן, עם המילה סופר נקרא גם לבנאי של מחלקת האב בשורה הראשונה בלבד! תכונת נוספת מעניינת - ניתן להשתמש במתודה הקודמת לפני שדרסנו אותה באמצעות המילה סופר: אנחנו במחלקת הבן דורסים מתודה שקיימת במחלקת האב, נוכל להחליט שאנחנו קוראים לפעולה הקודמת מתוך הפעולה החדשה שדרסנו. למשל -

```
public void drawOn(drawface d)
{
super.drawon(d);}

```

## המילה השמורה :Protected

תכונה שנוסיף במקום *private*. במחלקות שירשות - ניתן להשתמש באופן ישיר באמצעותה בתכונות של מחלקת האב. אך! אם ננסה להשתמש בו במחלקות שאינן יורשות ממנו הרי שזו שגיאה.

### חשוב מתרגול 3 (צביקה ברגר)

אוספים - *collections*:  
- יש שני סוגי אוספים. *list* ו *set* כאשר בסט אין חשיבות לסדר האיברים (קבוצה), נתמקד כעת בעיקר בליסט. יש שני סוגים -  
1. *ArrayList*  
2. *LinkedList*  
יש בהם הרבה מתודות בנויות שימושיות....

### הרצאה 4:

### המחלקה *class object*

מחלקה מובנית בג'אווה. שתי פעולות שראינו בעבר - *toString* ו *equals* שייכות לה. באופן אוטומטי ע"י ג'אווה, קלאס אובג'קט הופכת למחלקת האב של כל מי שלא יורש. עבור מחלקות שכן יורשות זה *class sub*.  
ניתן ליצור אובייקט של מחלקה אובג'קט - *Object o = new Object()*  
\* *getClass*: מתודה מיוחדת של המחלקה. מחזירה אובייקט שמתאר את הקלאס. ניתן להפעילה על כל אובייקט בג'אווה  
\* *hashCode*: מתודה של המחלקה שמחזירה ת"ז של אובייקט.  
כפי שהצגנו בעבר - יש הרבה בעיות עם *toString* ו *equals* ולכן נדרוס אותם לרוב ונשכתבם מחדש.  
נתבונן בקוד הבא שילווה אותנו:

```
public boolean equals(object other)
{
    return x==other.x && y==other.y;
}
```

במצב זה אכן הפעולה נראית חוקית - כלומר ניתן לקרוא איקס ולא *getX* כי זה מתוך המחלקה. אז מה כן הבעיה בקוד?  
איך נדע שאכן *object* הוא *point* ומחזיק ערכי איקס וואי?  
=====

### הפעולה *InstanceOf*

דוגמה תמחיש הכי טוב -

```
string s = new string("hello")
sop(s instanceof String) //==> True!
```

כלומר מה קורה כאן? הפעולה שואלת, האם *s* הוא אב קדמון של שם המחלקה מימין? כאן התשובה היא אכן כן.  
מעולה - פתרנו את הבעיה! נשנה מימוש ל -

```
public boolean equals(object other)
{
    if (!(other instanceof Point))
        return false;
    return x==other.x && y==other.y;
}
```

ובכן לצערנו גם זה לא מתאים - ומה אם הטיפוס שלנו *other* הוא אב קדמון של פוינט? נוכל להשוות בין *point* לבין איזו "*smartPoint*" <== לא טוב!  
מה הפתרון?



נוסיף -

```
if(!other.getClass().getName().equals(this.getClass().getName()))
וכעת טיפלנו בבעיה. כלומר מה למדנו? ניתן לגשת עם getClass גם לשדות עצמן שהאובייקט
מתאר, כמו שם. וכעת כמובן שאם שם המחלקה שונה <== לא אותו טיפוס וסיימנו.
```

### נושא 3 - Static&Dinamic bidding

דוגמה הכי טובה -

```
Polygon n = new name();
הפוליגון - הוא הטיפוס הסטטי! וname הוא הטיפוס הדינאמי.
בזמן ריצה קומפיילר בונה טבלת vtable לכל class בזכרון, טבלה שמחזיקה חלק(!!!) של הכתובות
של המתודות של המחלקה.
*אם מישהו ירש - הכתובת למתודה תהיה היכן שהכתובת למתודה של האבא.
קומפיילר דוחף שדה לכל אובייקט של vptr שיתאר את הכתובת בזכרון.
בואו נתבונן בדוגמה של דינאמיק בינדינג -
```

```
Polygon t = new SpottedTriangle(..)
t.getName();
מה שיקרה במצב כזה הוא דינאמיק בינדינג. כאן הוא יבצע את getName של spottedTriangle
```

#### לסיכום -

**סטטיק בינדינג** - מתרחש בזמן קומפילציה - כלומר, בשלב הקומפילציה, הקומפיילר יודע בדיוק איזו מתודה יש להפעיל. יותר משתלם בזמן הריצה.

**דינאמיק בינדינג** - מתרחש בזמן ריצה - כאן הקומפיילר לא יכול לדעת מראש איזו מתודה תופעל, ובמהלך הריצה הקוד יבחר איזו גרסה של המתודה להפעיל, בהתאם לסוג האובייקט שבפועל נמצא בזיכרון. זה קורה בעיקר עם מתודות שנמצאות בירושה. אם ניתן להמנע ממנו, מומלץ כי זה יחסוך זמן ריצה.

\* שלושה סוגי מתודות לא נכנסות ל*vptr*. - פרייבט, סטטיק ופינל.

זמן טוב לדבר על **מתודות פינל** - *final version*, לא ניתן לדרוס אותה ואם ננסה נקבל שגיאת קומפילציה. באופן דומה, *class final* היא מחלקה שלא ניתן לרשת ממנה.

\*\*\*\*\*

\* אעיר כי יש דוגמה מצויינת במצגת 4 על הנושא בעמוד 31.

\* עבור מתודה סטטית, בקומפילציה זה ילך לטיפוס הסטטי ולא הדינמי.

\* עבור משתני המחלקה - תמיד ניגש עם סטטיק בינדינג.

### מתי ניתן לעשות *Override*?

כללים -

1. חתימה זהה, שם המתודה סדר וטיפוס ארגומנטים (שם לא רלוונטי של משתנים)
2. ערך ההחזרה במתודה הדורסת מאותו טיפוס / טיפוס ספציפי יותר
3. נראות המתודה הדורסת זהה / גדולה יותר

### העמסה - *Overloading*

מתודות בשמות זהים, השוני - הארגומנטים ששלחנו.

מה אסור? מתודות באותו השם, פרמטרים זהים, השינוי הוא בערך ההחזרה - לא חוקי ולא מתקמפל. *ambiguous*

כמו כן גם שוני בשמות ארגומנטים לא נחשב, אלא רק מס' שלהם / ערך טיפוס שלהם

\* דוגמה חשובה -

יש שתי מחלקות A ו B. B מרחיבה את A.

```
func(A x)
func(B y)
```

מה קורה במצב זה? קומפיילר בוחר לפי סטטיק בידינג, בהתאם לטיפוס הסטטי של ארגומנטים כפי שהסברנו. לעומת זאת,

```
func(A x, B y)
func (B y, A x)
```

כאן נקבל שגיאת קומפילציה.

## הרצאה 5:

### העמסת בנאים (*constructors of overloading*):

\* אם הגדרנו מחלקה ללא בנאים, יהיה בנאי דיפולטי. \*\*כשנבנה אח"כ מופע של המחלקה שיש בה בנאי דיפולטי כמובן שלא נשלח פרמטרים. מדוע בכלל נרצה זאת? לא תמיד נרצה לחייב את המשתמש לשלוח את כל הפרמטרים בבניית המופע. (למשל - לא כולם במחלקה *Student* ירצו להעביר מידע על תאריך הלידה שלהם או על המין שלהם) ניתן שהמשתמש ישלח *null* בפרמטר שהוא לא מעוניין לשלוח ערך אבל בדיוק בשביל זה יש לנו העמסת בנאים - שזה יראה אלגנטי יותר. \*\* כשנעשה העמסת בנאים - לא נרצה שיהיה שכפול קוד בשני בנאים שונים, ולכן ניתן לקרוא לבנאי אחד מבנאי אחר. איך? נעשה *this* וזה ניתן לעשות רק בתוך קונסטרוקטור: למשל -

```
public - Student(string - name, string - birthday){
```

```
    this.name = name;
```

```
    this.birthday = birthday; }
```

```
public - Student(stringname){
```

```
    this(name, null); //--- >
```

בשורה השנייה קראנו לבנאי אחר של המחלקה. בעצם בבנאי השני החלטנו שנרצה לשלוח רק שם ללא תאריך לידה. \*\*מותר לעשות כמה בנאים שנרצה.

## מה קורה מאחורי הקלעים בהפעלת קונסטרקטור?

1. ראשית מקצים זכרון עבור כל השדות של האובייקט (כולל *Vtable* ולמחלקת *object*).
  2. אחר כך נקצה ערך דיפולטי לכל אחד מהשדות של המופע.
  3. אם בתוך קונסטרקטור היה קריאה לקונסטרקטור אחר - נבצע קודם אותו.
  4. גם אם כתוב וגם אם לא כתוב נצטרך ללכת לטפל ב-*super*. (אם יש *super* נלך לעשות אותו) ואם לא כתוב שום דבר עדיין אנחנו מחויבים ללכת לבנאי של מחלקת האב הגדול - *object*. כלומר תמיד תתבצע קריאה ל-*object* בהפעלת קונסטרקטור (גם אם לא ראינו זאת באופן מפורש בקוד) - גם אם קראנו לבנאי אחר הוא יהיה אחראי לקרוא ל-*object*.
  5. להמשיך לבצע את "גוף" הקונסטרקטור הנוכחי. סיימנו לבנות את הפרויקט.
- \*\*reference** זה 8 בתים, *string* הוא רפרנס. כמו כן נזכר כי *int* הוא 4 בתים. כאשר מקצים אובייקט בזכרון למשל של *student* עם *age* ושני *strings* - יהיה לנו  $4 + 2 * 8$  בתים ועוד 8 בתים למצביע של *Vtable* שתמיד יש מאחורי הקלעים + עוד בתים שיש בשדות של מחלקת *Object* (נדע בהמשך).. - < בקיצור מתחבא הרבה מאחורי הקלעים.

### דוגמה טובה: נניח ויש לי מחלקה ריקה

*public class C*  
 נרצה להפעיל עליה המרה לטיפוס *B* כנל - *((B).c).g()* נקבל שגיאת קומפילציה. כיוון שאין שום קשר בין *B* ל-*C* הקומפיילר לא יתן לנו ונקבל שגיאת קומפילציה.  
**דוגמה נוספת:**

```
public class A {
    public void f()...}
    public class B extends A { f()... g()...}
```

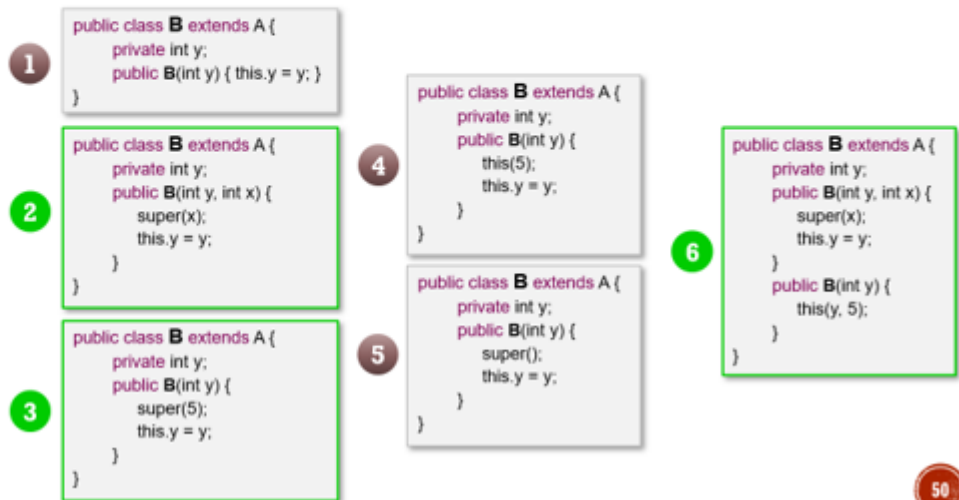
ונניח ויש לנו טיפוס *a* מסוג *A* ונרצה להריץ את השורה הבאה -  
*((B).a).g()* - קומפיילר יאשר כיוון שעל נייר יתכן שהם אכן מאותו טיפוס אך נקבל שגיאת זמן ריצה כי בסוף *a* מטיפוס *A*.

## SANITY TEST



```
public class A {
    private int x;
    public A(int x) { this.x = x; }
}
```

בהינתן הגדרה של *class A*, סמנו את כל ההגדרות התקינות של *class B*.



דוגמה 1: שגויה כי חייב להיות קריאה ל-*super* ואין כזו - כלומר הקומפיילר מוסיף כזו ללא ארגומנטים וזו שגיאת קומפילציה כי לא קיים לא *A* בנאי ללא ארגומנטים.

דוגמה 2: חוקי כמובן.

דוגמה 3: נשים לב שחוקי כיוון שמותר בהורשה שהיו שני קונסטרקטורים שמקבלים אותו מס' ארגומנטים.

דוגמה 4: לא חוקי כיוון ש-*this* קורא באופן רקורסיבי לכאורה לעצמו - ולפי *java* לבנאי אסור לקרוא לעצמו, וכן גם אין קריאה ב-4 ל-*super*.

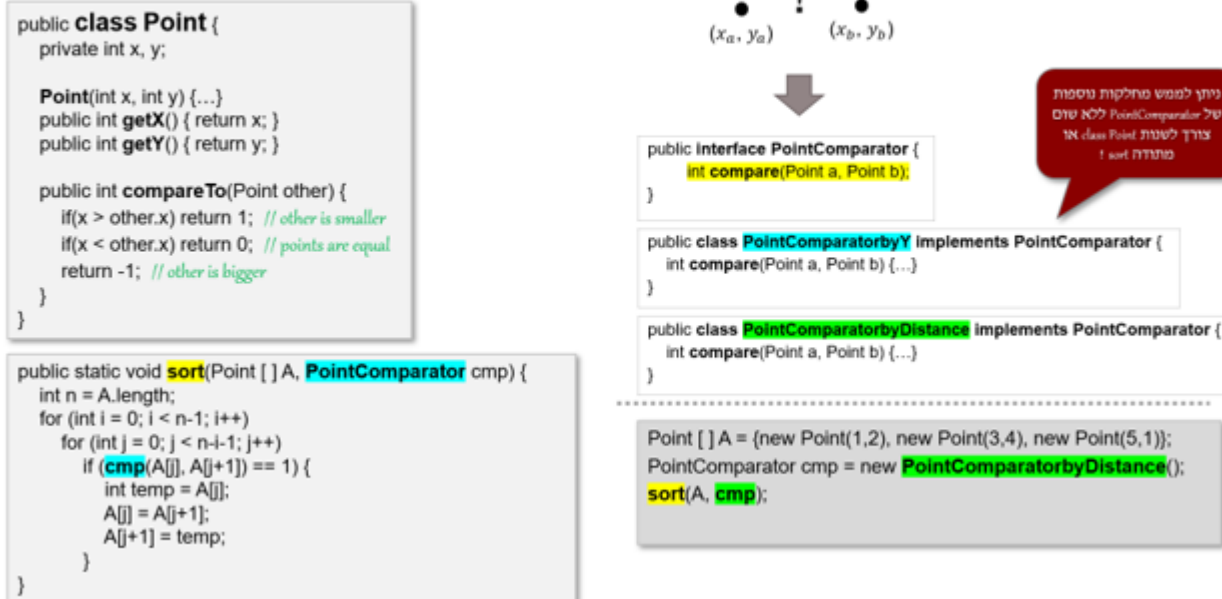
דוגמה 5: לא נכון כיוון שכמו ב-1 קראנו ל-*super* ונקבל שגיאת קומפילציה כי אין בנאי ריק ב-*A*.

דוגמה 6: תקין כמובן.

## :Generics&collections

עד 2004 לא היה קיים ב-*java*.  
 מוטיבציה - אנחנו במחלקה *point* רוצים לקבל מערך נקודות וליצור מתודה שתמין אותם. לפי מה? שאלה מעולה - לפי מרחק מראשית הצירים, לפי מרחק מציר *x*, ... יש הרבה אפשרויות. באופן עקרוני על הנייר ניתן לכתוב מתודות שונות לכל סוג מיון - אך יש כאן שכפול קוד וזה לא נכון ב-*java*!  
 \*\*הערה: בנוסף ל-*toString*, *equals* עלינו מעתה לממש גם את המתודה *compareTo* מחדש. בדומה ל-*C*, נוכל ליצור פונקציות גנריות. כיצד נממש?

## GENERICS



\* ניצור ממשק *interface* שיחזיק במתודה הכללית *compare*. ואז אנחנו נחליט לשלוח בכל פעם השוואה אחרת ל-*PointComparator* שעל פיה תעבוד ההשוואה. נשים לב שכאן כל סוג מיון הוא מחלקה שונה!

אך כפי שיש לנו השוואה בין שני *points* נרצה לעשות השוואה בין שני קלפים למשל - אבל מה, שוב נעשה את הממשק וכל המחלקות האלו?! כמה ממשקים אפשר ליצור... זה נוגד את העקרונות של *java* שאוסרים על שכפול קוד (גם אם הוא רק זהה ולא אותו דבר)

יש ממשק *Comparator* שכבר נמצא בספרייה הבסיסית של *java*.

```
point interface Comperator <T>{  
int compare ( T a, T b);
```

מדובר בכלי של *java* שנקרא *generic*.

כשנממש את ה *interface* אנחנו נכתוב במקום *T* את הטיפוס שנרצה, למשל -

```
public class CCByStrength implements Comperator <Card>{  
int compare (Card a, Card b) return a.getRank()-b.getRank();
```

}

ואיך נראית הקריאה?

```
CCbyStrengthcmp = newCCbyStrength();  
cmp.compare(newCard(8,"red","h"),newCard(10,"red","h"));  
מה היתרון של השיטה הזו? כאן - נשתמש ב interface אחד בלבד!
```

:*Generic Collections*

אוספים גנריים מאפשרים שימוש בתבנית יחידה לאוסף כלשהו. נרצה ליצור אוסף בלי לציין את שם הטיפוס. למשל ביצירת מערך אנחנו מחויבים לציין את הטיפוס.

ולכן הגיע הנושא לעולם - יש הרבה מאוד אוספים גנריים בג'אווה... *List < E >* וכן *Set < E >* וכן *Map < E >* וכן *Queue < E >* ועוד...

כשנרצה ליצור אוסף מסוים אנחנו מתחייבים כבר לטיפוס מסוים! מתחייבים בפני קומפילר ואם אח"כ ננסה להפר את זה נקבל שגיאת קומפילציה -

```
LinkedList < Student > l = newLinkedList <> ();  
l.add(newStudent("Arbel",1));
```

\* שאלה - מדוע אנחנו מסתבכים ולא מגדירים מחלקה כדקלמן:

```
publicclassArrayList  
private - Object[]elements = newObject[10];privateintsize = 0;
```

מה הבעיה כאן? הרבה - 1. אכן ניתן לדחוף לאוסף הזה מה שנרצה מבחינת דטה טייפ, אבל אפשר לדחוף הרבה טיפוסים שונים(זה לא פסול) אך זה קשה כי צריך לזכור כל פעם סוגי דטה טייפ, ויש חשש לקבל שגיאת זמן ריצה. כלומר - ניתן להשתמש בזה אבל יש להזהר גם בהוצאה וגם בהכנסת איברים.

\*\* ניתן להגדיר כך - *privateClass <? > allowedClass;* מה זה אומר? ניתן להצביע לכל מחלקה מהסימן שאלה, זה סוג של דטה-טייפ *Class <? >*

:*File - IO*

ניתן לעבוד גם עם קבצים בדומה ל *input*.

*streams* - זרם של נתונים ממקום למקום (לא שונה מזרם של מים...). פתחנו קובץ וכעת ניתן לקרוא ממנו תו תו (באופן דומה לטיפות המים.... (יואב ביקש שאני אכתוב )) קריאת נתונים -

```
opening out stream while(can read) :  
do something...  
close stream.
```

בכל שפה שהיא ככה עובדים עם קבצים.

דוגמה פשוטה בג'אווה -

```
Filefromfile = newfile("a.txt")  
fileFileInputStreamfis = newFileInputStream(fromfile)  
fis.read();...fis.close();
```

אנחנו עובדים על האובייקט שמקושר לקובץ ולא על הקובץ עצמו.

כשנעשה `fis.read()` אנחנו מושכים יחידה שמורכבת מ-`byte` של הקובץ, משכנו `byte` ולא תו! \*\*טוב לדעת כי בג'אווה ניתן להרחיב את טבלת האסקי ולייצג גם אימוג'ים וכו'.... בג'אווה `char` זה 2 בתים.

\* כאשר נגמרת הקריאה `fis.read` יחזיר -1.  
 \* בשביל לכתוב את הנאמר בהדפסה יש פקודה מיוחדת - `system.out.write(readByte);` - מה פקודה זו עושה? `systemOut` הוא `stream` לכל דבר ונשתמש בה בשביל לכתוב את הבייט שנכתב - היא מדפיסה רק את הבייט האחרון לא כל הארבעה בתוך אינט - היא טיפשה ופרמיטיבית.  
 \* נוסף לקוד במיין בכותרת - `throws IOException` - נבין בהמשך מה זה אומר.  
 \* ניתן לכתוב לתוך קובץ - באמצעות `FileOutputStream`.  
 \* תמיד נסגור את ה-`stream` שפתחנו - אחרת יהיה `memoryLeak`.  
 \* `BufferedReader`: הבאפר עובד עד שהוא קולט בקסלאש אן. הוא מעתיק בייט בייט לתוכו עד שפוגש בקסלאש אן ואז ניתן להדפיס את השורה שהוא קרא והוא מרוקן כל מה שהיה בתוכו.

## תרגול 5

\* עקרון ה-`encapsulation`: עקרונו הוא להחביא חלק מהמידע של האובייקט מהמשתמש. כלומר הלקוח לא צריך לדעת בין היתר כיצד מימשנו את הדברים ובאיזה שדות השתמשנו.  
 מאפשר למנוע מהמשתמש לשנות את המידע כיצד שהוא רוצה, אני אחליט מה הוא יוכל לעשות. כלומר אני אומר למשתמש - תעשה ככה, אני אדאג שזה יעבוד (למשל פקודת `printf`)  
 מודולריות - אנחנו יכולים לעשות קוד עם תחזוקה ברמה גבוהה. מאפשר החבאה של מידע.  
 \*כמובן שאסור לגשת למשתנים פרטיים ואם ננסה נקבל שגיאה.  
 עקרון ה-`Refactoring`: בנייה מחדש של קוד קיים שיהפוך לקל יותר לתחזוקה, גמיש ותקין.  
 במקום לעשות `else if` מלא פעמים שדי חוזרים על עצמם - נשתמש בעקרון זה. נגדיר מחלקה בהתאם לכל מקרה פרטי...  
 \* הערה חשובה (זה ממש הופיע כשאלה במבחן): לא יתכן שיש לנו מתודה שהיא `static abstract` - מדוע? מתודה אבסטרקטית היא כזו שניתן לדרוס אותה ולממש מבחוץ, מתודה סטטית היא של המחלקה ולא ניתן לדרוס אותה - לכן סה"כ לא יתכן אחת שהיא גם וגם!  
 ---

## הרצאה 6 - Exceptions:

חריגה ודברים יוצאי דופן. הגדרנו מושג בג'אווה שהוא תקין אך לא נכון - אנחנו כעת נדבר על מצב של לא תקין.  
 \*יש אפשרות בג'אווה לבדוק האם קובץ קיים. אוקיי בדקתי והוא קיים - מי אמר שרגע אחרי הוא קיים? אולי מישהו מחק אותו באותו זמן? נחלק את הנושא לשניים - בעיות בשליטתנו ובעיות שלא בשליטתנו - בכל מקרה זה יחשב לא תקין. עלינו להתמודד עם כך בשני המצבים!  
 \*מערכת ההפעלה תהיה המעורבת והיא תהיה הראשונה שתדע על הבעיה. בין אם זו בעיה של חילוק באפס ובין קובץ שלא קיים. מערכת ההפעלה שולחת `JVM signal` - זה איתות ל-`jvm` - תעשה משהו עם הסיגנל אחרת התוכנית תקרוס. ה-`JVM` מייצגת אובייקט מיוחד - האובייקט מסוג `error` והוא "נזרק" (ככדור) - התוכנית שלנו יכולה לתפוס את האובייקט הזה - לתשאל אותו - ובהתאם לכך לטפל בבעיה שנוצרה.  
 דוגמה - ג'אווה אומרת לנו שימו `try` - נסו לעשות את מה שאתם רוצים, ואז תגדירו בלוק של `catch` - אם יגיע כדור של "error" - אנחנו נתפוס אותו. מדובר בשגיאות של קובץ לא נמצא, קובץ נמחק וכו'.... יש לשים לב כי `try` ו-`catch` אינן מתודות - הן חלק סינטטי מהשפה - כמו `if`.

```

public static void readFile() {
    BufferedReader reader = null;

    try {
        reader = new BufferedReader(new FileReader("a.txt"));
        String s;
        while((s = reader.readLine()) != null)
            System.out.println(s);
    } catch(IOException e) {
        ... //handle the IOException
    }
}

```

כדי לתפוס חריג, עלינו לעטוף את הקוד שעלול לזרוק חריג ב- try block, ולהוסיף catch block לעבור החריג הצפוי

BufferedReader constructor throws IOException if an IO error occurs while opening the input file

public String readLine() throws IOException {...}

public int read() throws IOException {...}

\* נשים לב כי מצטרף לחתימת המתודה שנרצה שתשלח שגיאה throws exception מסקנה - ההתנהגות הזו תעזור לנו למנוע מהתוכנית לקרוס. היא תשלח הודעת שגיאה במקום להקריס את התוכנית.

\* ניתן לשים כמה catch שנרצה - זה בדיוק switchCase. כמו כן נזכר כי נלך לcatch הראשון שיופיע - לכן נרצה לשים את ההכי ספציפי למעלה ובסוף את ההכי פחות ספציפי (הכללי יותר).

\* יש מחלקה שיורשת מIOException - FileNotFoundException. נרצה לטפל במצב זה שונה כאשר הקובץ לא נמצא. (אם כי יש תמיכה לכך גם במחלקה המקורית). ל-e יש מתודה שהיא GetMessage שניתן להדפיסה ולהבין מה הבעיה - וכך לחלק למקרים, כמו בדוגמה כאן -

```

public static void readFile() {
    BufferedReader reader = null;

    try {
        reader = new BufferedReader(new FileReader("a.txt"));
        String s;
        while((s = reader.readLine()) != null)
            System.out.println(s);
    } catch(FileNotFoundException e) {
        String msg = e.getMessage();
        if (msg.contains("Permission denied"))
            System.out.println("permission denied");
        if(msg.contains("No such file"))
            System.out.println("File does not exist.");
    } catch(IOException e) {
        ... //handle the IOException
    }
}

```

\* נשים לב שעלינו תמיד לסגור את stream. כיצד נסגור אותו בתוך catch? הרי אם נסגור בסוף הראשון לא נוכל להגיע קדימה. בג'אווה נפעיל טרי אנד קץ' בכל פעם על reader.close() אבל יהיה כפילות קוד וזה לא לפי עקרונות oop. לשם כך בסוף כל tryCatch נוסיף finally - זה בלוק של הקוד שמתבצע בין אם קרתה השגיאה ובין אם הכל היה תקין. הוא רץ מיד אחרי הביצוע בcatch שתפס את השגיאה, נשים שם קוד שיתבצע בכל מקרה למשל סגירת streams. יש מקרה אחד בדיוק שfinally לא מתבצע - אם יש פקודת exit באחד הcatch לפני.

כך זה נראה בקוד -

```
public static void readFile() {
    BufferedReader reader = null;

    try {
        reader = new BufferedReader(new FileReader("a.txt"));
        String s;
        while((s = reader.readLine()) != null)
            System.out.println(s);
    } catch (FileNotFoundException e) {
        ... //handle the FileNotFoundException
    } catch (IOException e) {
        ... //handle the IOException
    } finally {
        if (reader != null)
            try {
                reader.close();
            } catch (IOException e) { ... }
    }
}
```

ותבצע בין אם קרתה  
אין. הוא רץ מייד אחרי  
תפס את השגיאה. ב-  
ע בכל מקרה, למשל,  
וקרה היחיד בו *finally*  
קודת *out* או בקוד של  
שהקפצנו אליו.

- *err* הוא גם *output* של *System.err.stream* הוא אוטופוט מידי של השגיאה (באופן שמוכר לנו זה ידפיס למסך אך זה רק דיפולטי - יש אפשרות להדפיס את זה במקום אחר) - אם ננסה להדפיס באמצעותו משהו על המסך היא תקבל עדיפות עליונה - שלא כמו *system.out* - ולכן כאשר נרצה להדפיס באופן מידי נכתוב את השורה הבאה -

```
system.err.println(e.getMessage());
```

משתמשים בתכונה זו רק להדפסת שגיאות.

עלינו לדעת שבכל שפה יש שלושה *streams*:

1. *InputStream* - מקלידים למחשב.

2. *OutputStream* - רגיל הדפסה.

3. *OutputErr* - לשגיאות.

ישנו מושג שנקרא *activationFrame* - זה המקום שבו יש שמירה של הזכרון של כל המתודות. ניתן להדפיס למען דיבוג *stackTrace* שידפיס את כל המתודות מלמטה למעלה בסדר הפעלתן ונבין היכן יש בעיות. כלומר *main* יודפס הכי למטה.

מתודות שזורקות *exception*: מי שקרא למתודה כזו הוא חייב לטפל בשגיאה וליצור לה *tryCatch* ושם בקריאה למתודה יטפלו בזה ואם לא יטפל בזה תהיה שגיאת קומפילציה. במתודה עצמה לא יהיה טיפול וזריקה של אקספסן. האם זה בכלל עדיף על השיטה שראינו קודם? כן - לטפל בשגיאה עדיף לעשות היכן שקראנו למתודה, שכן הטיפול הוא פרטני. ולכן כל המתודות של *tryCatch* בג'אווה לא יזרקו שגיאה - האחריות היא על מי שקרא למתודה זו הדרך הנכונה וכך נפעל בג'אווה.

ישנה בעיה - במידה ויזרק שגיאה לא נגיע לשורת ה*close* כי בזמן ה*try* נקבל שגיאה ולכן לא נחזור למתודה בכלל ולא נוכל לסגור את ה*stream*! מדוע לא נוכל לסגור את הקובץ ב*main*? הרפרנס לקובץ הוא לוקלי של המתודה שקראה לו - ולכן בחוץ ב*main* הקובץ לא מוכר. המימוש לפתרון הבעיה יהיה כדקלמן -

```
public static void readFile() throws IOException {
    try{
        BufferedReader reader = null;
        reader = new BufferedReader(new FileReader("a.txt"));
        ... }
    finally{
        try{
            reader.close();
        } catch (IOException e) { ... } } }
```



מה קרה כאן ? הייתה קריאה כלשהי בדרך מה *main* של *catch* - לאחר *catch* תמיד הולכים ל*finally* ונלך ל*finally* שבתוך המתודה, ונוכל לסגור את הקובץ.  
באופן שקול את אותה הבעיה ניתן לפתור באופן קצר יותר ויזואלית (אך מאחורי הקלעים קורה אותו דבר) כך -

```
try(BufferedReader reader= new BufferedReader(new FileReader("a.txt"))
לשיטה זו קוראים tryWithResource - ימנע מאיתנו לכתוב finally לחינם וזה יעשה את זה
במקום מאחורי הקלעים.
נשתמש מעכשיו רק בזה לסגור קובץ - כאשר נשים בתוך סוגריים של try הם יסגרו אוטומטית
לאחר מכן - מבלי שנצטרך לעשות finally כמו בדוגמה למעלה.
* יש לשים לב - נניח ויש לי f ששולחת שגיאה ויש g שקראנו בתוכה לf - עבור g יש שתי
אפשרויות: או להגדיר בתוכה tryCatch או להגדירה כשולחת שגיאה.
```

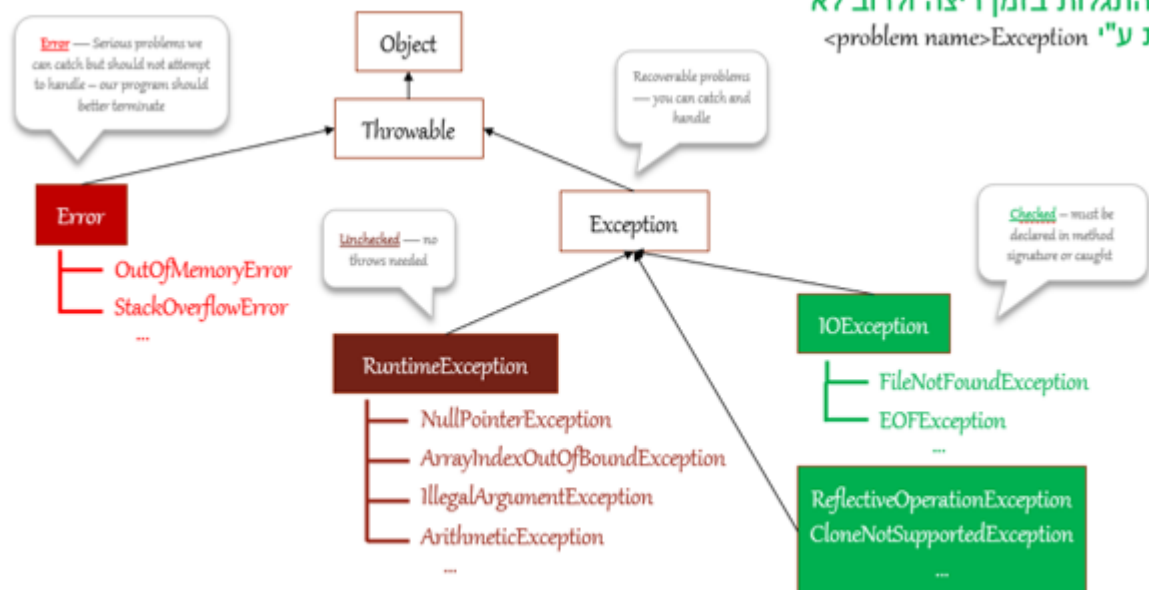
## סוגי שגיאות:

יש סוג בעיות שניתן לתפוס כמו כאן -

```
int x, y; ...
try { // code that may throw a ArithmeticException
    n x = x / y; }
catch (ArithmeticException e)
{ // handle the exception here System.out.println("I caught it!"); }
```

ניתן לתפוס אך זה לא נכון! זה באג ואחריות המתכנת היא לטפל בו!

היררכית השגיאות בג'אווה  
נלהתגלות בזמן ריצה ולרוב לא  
גות ע"י <problem name>Exception



*Error* - דבר שניתן לתפיסה בדר"כ. לעומתו *Exception* מתחלק ל-1. זה העולם הירוק - הקומפילר מחייב אותי לתפוס אותם בעצמי אחרת הוא יכעס מאוד ולא יתן לקמפל. לעומת זאת יש את העולם האדום - אלו שגיאות שהן שגיאות זמן ריצה - כל הבאגים.. חלוקה באפס, גישה לאינדקס מחוץ לגבולות מערך וכו'... אותם קומפילר לא מחייב לתפוס בעצמי.  
דוגמה ל*catch* של *error* - אפשרי כדי להדפיס את השגיאה ואת *stacktrace*, ומומלץ מייד לסיים את ריצת התוכנית.

דוגמה: מה יקרה כאן? קיבלנו  $i = 2$  ונצטרך לשלוח חזרה את המתודה שלמעלה - שהיא כמובן מדפיסה *UnexpectedValue*. הערה - כאן כיוון שיש הרחבה של המתודה *runTime* ששם אין חובה לתפוס את הבעיה - לא היינו חייבים לעשות כאן *tryCatch*.

```

public class UnexpectedOption extends
RuntimeException {
    public String getMessage() {
        return "Unexpected value\n";
    }
}

public static void func(int i) throws
UnexpectedOption{
    if(i == 2)
        throw new UnexpectedOption();
    ...
}

public static void main(String [ ] args) {
    try {
        func(2);
    } catch (UnexpectedOption e) {
        System.err.println(e.getMessage());
    }
}

```

#### דוגמה נוספת:

```

public class Main { public static void readFile() throws IOException { throw new IOException(); }
public static void main(String[] args) throws IOException { readFile(); } }

```

מה קורה כאן? בג'אווה החליטו ש *main* יכול לזרוק *exception* - מי שנמצא מעל ה *main* הוא *JVM* - הוא תפס את השגיאה ולכן התוכנית תקרוס!

### הערות חשובות מתרגול 6:

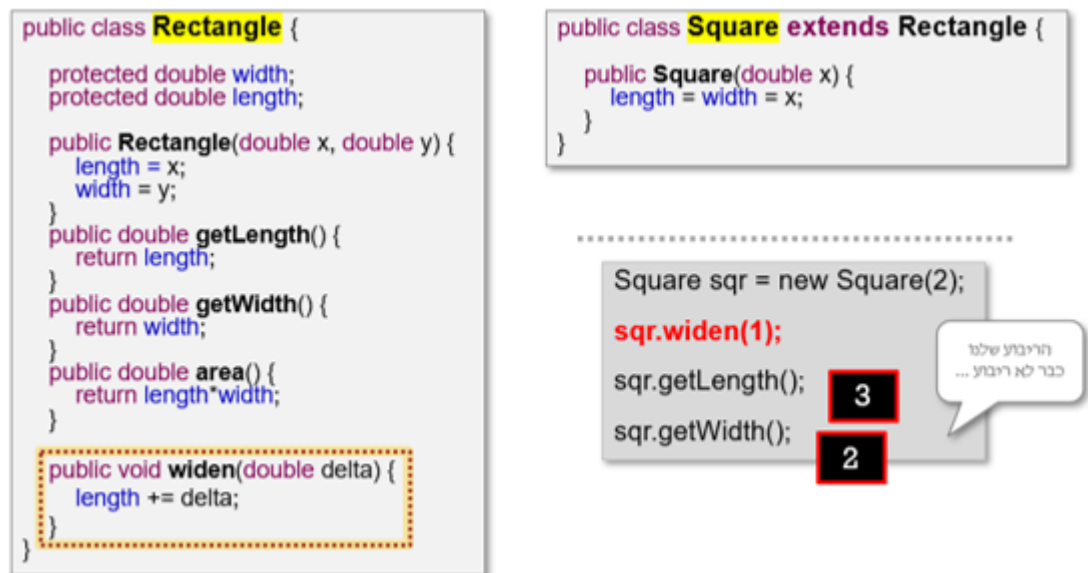
- \* כל סוגי החרigות יורשים מהמחלקה *exception*.
- \* *check* - על המשתמש לטפל בשגיאות. *unchecked* - לא מחוייבים בקוד לטפל בהם. *error* - קיים, לא נתעסק - למשל: מחשב קרס, אין מספיק זכרון במחשב.
- \* *StackTrace* - הודעה על המסך שתוצא כאשר יש שגיאה, היא שולחת "קישורים" והפניות לשורה בקוד בה נוצרה החרigה.
- \* הקשתנים יכולים להרחיב מאוד - במקור אנחנו קוראים בייט בייט אח"כ הבנו שנוכל לקרוא *char* ובהמשך הבנו שנוכל לקרוא *line*. חשוב לזכור שבפועל כשמפעילים את הבאפר רידר מאחורי הקלעים נקרא בייט בייט.
- \* יש הרבה הרחבות לטבלת אסקי - למשל *utf8* שמאפשר שימוש באימוג'יז עברית סינית וכו'.... זה כבר לא ברמת הבייט.

### הרצאות 7 - *Design – patterns*:

מוטיבציה: מה זה עיצוב? תכנון נכון של התוכנית שלי. מה זה תבנית עיצוב? יש איזשהי תבנית / הצעת עיצוב למצבים שכיחים באופן יחסי. יש פתרון שאנשים חשבו עליו וחסכו לנו הרבה - הם יצרו תבניות עיצוב למצבים שכיחים. כמה נלמד? 5-6. יש הרבה הרבה יותר...

הערה: מופיע בסוף הרצאה 8 כאן בסיכום סיכום של תבניות העיצוב - מומלץ. יותר יעיל מלקרוא מה שכתוב כאן שכן יש בסה"כ להבין את הרעיון.

נדמיין מצב בו נקח סטודנט שכבר עשה את הקורס - מרינה תקח אותו ותשים אותו מאחוריה, תשאל את כולם שאלה ותתן לו לענות - מרינה נתנה לו האצלה - ברור שהוא ידע את התשובה. זה כלי של OOP שמתחרה מאוד עם הורשה - זו חלופה טובה להורשה. (לסיטואציות מסוימות). דוגמה יותר רלוונטית: יש פרויקט עם ריבוע ומלבן - כיצד נממש? כל ריבוע הוא בפרט מלבן - לכן ניצור מחלקה שתקרא מלבן וניצור מחלקה שתורש ממלבן והיא תהיה ריבוע - מדוע? כל ריבוע הוא מלבן. פשוט נגדיר שבמחלקה ריבוע מתקיים  $height = width$ . על פניו, זה נראה טוב - זה לא! מדוע? לפי עקרון ההורשה - אם B מרחיבה את A - צריך שבכל הקשר בו משתמשים בטיפוס B נוכל להשתמש גם ב-A, המתודה הבעייתית היא `widen` - כשנפעיל `widen` על ריבוע נקבל שהריבוע - יהפוך למלבן! וזה לא טוב! עם זאת - יש למחלקות הרבה קוד חופף. איך ננצל זאת? לשם כך מגיע העקרון שלנו - תמיד נעשה `extend` אם כל מה שמוגדר עכשיו מתאים לי ואם כל מה שמוגדר לי בעתיד יותאם לי. כאן כמובן זה ממש לא המצב.



(הערה - יש שגיאה במצגת של מרינה צריך להיות `super` בתוך הקונסטרוקטור של ריבוע) אז מה הפתרון? ממש לא נשכפל קוד - לשם כך בא `deligation`. ריבוע כאן לא תרחיב את מלבן - אלא מלבן יהפוך להיות שדה של המתודה. בכל פעם אנחנו עושים "האצלה" - ועונים בשם `.delegate`.

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

    public Rectangle(double x, double y) {
        length = x;
        width = y;
    }

    public double getLength() {
        return length;
    }

    public double getWidth() {
        return width;
    }

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

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

```
public class Square extends Rectangle {
    private Rectangle delegate;

    public Square(double x) {
        delegate = new Rectangle(x, x);
    }

    public double getLength() {
        return delegate.getLength();
    }

    public double getWidth() {
        return delegate.getWidth();
    }

    public double area() {
        return delegate.area();
    }
}
```

במקום  
! ext



```
Square sqr = new Square(2);
```

הורשה של המתודה  
widen לא מופיעה על  
class Square

כעת - אם נרצה להפעיל על ריבוע מתודה *widen* - נקבל שגיאה - כיוון שמתודה זו באופן ישיר ממש לא נמצאת שם. כלומר - אפשר להפעיל את המתודות שנמצאות בלבד בתוך המחלקה שלי. כלומר כאן אנחנו נבחר מה לקבל - בניגוד להורשה.

דוגמה עם עץ בינארי ל-delegated

\*נרצה לדעת כמה הכנסות בוצעו לתוך עץ בינארי, כאשר אנחנו מכניסים ערך לעץ ה-*counter* גדל וכאשר נמחק מהעץ הוא לא ישתנה - נספור כמה הכנסות בוצעו סה"כ. על פניו - נוכל להרחיב את המחלקה שלנו למחלקה "עץ עם מונה" כאשר נוסיף שדה של *counter*. הורשה נראית מתאימה מאוד! נוסיף כמובן שינוי *add* עם *override* כאשר אנחנו נוסיף לעץ אנחנו נעדכן ++ *counter*. מה הבעיה?

```
BinaryTree<String> t = new BinaryTree<>();
t.add("a");
t.add("b");
t.add("c");
BinaryTree_withCounter<String> ct = new BinaryTree_withCounter<>();
ct.add("x");
ct.addAll(t);
System.out.println("Insertion count is:" + ct.getCounter());
```

ציפינו לקבל  
הדפסה של 4. מה  
בדיוק קרה כאן !!

נראה מעקוב  
אחרי הקוד הבין

Insertion count is: 7

```
public class BinaryTree_withCounter<T> extends BinaryTree<T> {
    private int counter = 0;

    public int getCounter() { return counter; }
    @Override
    public void add(T object) {
        counter++;
        super.add(object);
    }
    @Override
    public void addAll(Collection<T> other) {
        counter += other.size();
        super.addAll(other);
    }
}
```

```
public class BinaryTree<T> implements Collection<T> {
    void add(T object) {...}
    void addAll(Collection<T> other) {...}
    for (T item : other)
        add(item);
}
//
```

עדיף להשתמש ב-  
class ל-  
BinaryTree

```
BinaryTree<String> t = new BinaryTree<>();
t.add("a");
t.add("b");
t.add("c");
BinaryTree_withCounter<String> ct = new BinaryTree_withCounter<>();
ct.add("x");
ct.addAll(t);
System.out.println("Insertion count is:" + ct.getCounter());
```

Insertion count is: 7

10

מדוע קיבלנו 7? שלב לפני האחרון אכן  $counter = 4$ , אך יש קריאה של `super` בסוף `addAll`, בתוך `addAll` אנחנו נכנסים אל `add`. ל-`add` של מי? של העץ עם מונה! בגלל דינאמיק בידינג. במתודה מסוג זה של `void` הקריאה מתבצעת לפי דינאמיק בידינג. איך נפתור את הבעיה? פתרון אחד - אפשר למחוק את הקריאה של `addAll` בתוך עץ עם מונה. אבל המחלקות מאוד תלויות זו בזו. אם מחר משהו מגיע ומשנה את `addAll` אנחנו נדפקנו. נרצה למצוא פתרון אחר - עם האצלה! נוסיף שדה של עץ בינארי לתוך עץ עם מונה. נממש כך -

```
public class BinaryTree_withCounter<T> implements Collection<T> {
    private BinaryTree<T> delegate;
    private int counter = 0;

    public BinaryTree_withCounter() {
        delegate = new BinaryTree<T>();
    }
    public void add(T object) {
        counter++;
        delegate.add(object);
    }
    public void addAll(Collection<T> other) {
        counter += other.size();
        delegate.addAll(other);
    }
    public boolean isEmpty() {
        return delegate.isEmpty();
    }
    public int size() { return delegate.size(); }
    ...
}
```

נשתמש ב-  
delegation  
מלבד מניית הפריטים  
שהוכנסו, כל שאר  
המתודות מואצלות.

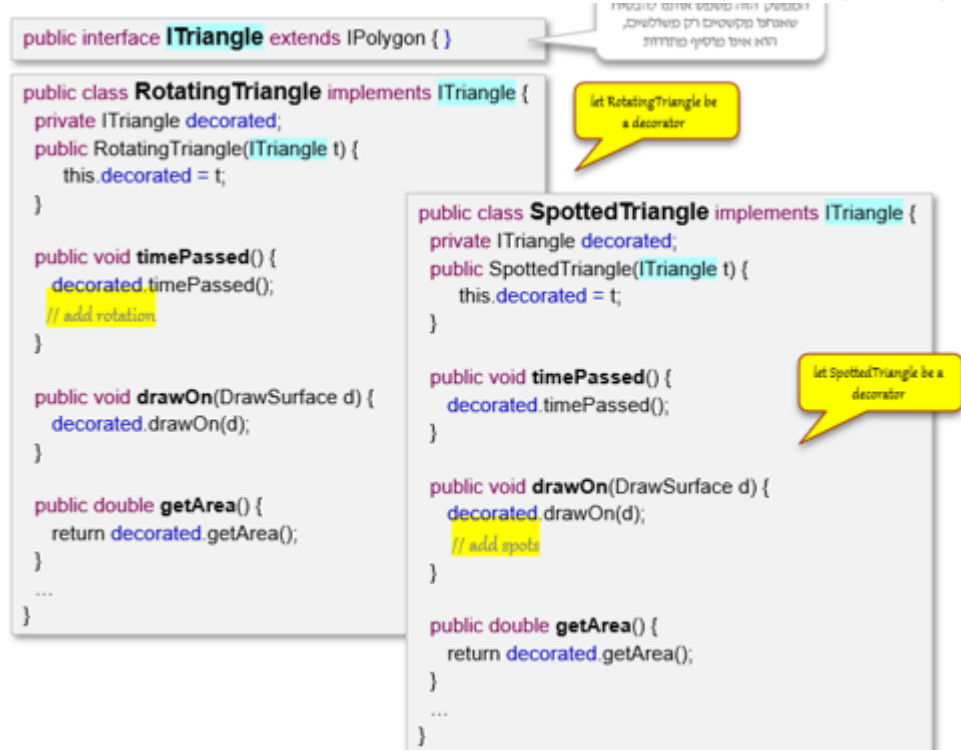
```
public class BinaryTree<T> implements Collection<T> {
    void add(T object) {...}
    void addAll(Collection<T> other) {...}
    for (T item : other)
        add(item);
}
//
```

נשאלת השאלה - מה ההבדל? הרי מימשנו כאן אותו דבר את `addAll`. הפעם אנחנו נשתמש ב-`add` של `binary - tree` ולא של המונה המורחב. מדוע? גם כאן יש דינמיק בידינג - אבל הפעם הוא הולך לפי סוג האובייקט. מי שלח אותי? `delegate` מסוג `binary - tree` ולכן שנעש ה-`add` אנחנו מסוג `binary - tree` ולכן נשתמש ב-`add` שלו הפעם! כתיבה כזו של מחלקה נקרא *pattern design decorated* - קשטון! הוספה של התנהגות או שדות תוך כדי שמירה על שדות והתנהגות מקוריים. הקישוט מתבצע ע"י האצלת כל הפקודות לאובייקט המקושט בתוספת שדות והתנהגות רצויים.

--  
**דוגמה חשובה:** אנחנו בפרויקט חשוב ורוצים לרשת משתי מחלקות שונות. למשל נעבוד עם

משולשים - יש לנו מחלקה של משולשים מסתובבים ויש מחלקה של משולשים עם נקודות. נרצה ליצור משולש מסתובב עם נקודות. מה נעשה? הרי אי אפשר להרחיב שתי מחלקות!! *delegate* **כמובן!**

נתחייב להיות *polygon* ולא נעשה *extends* בכלל. נממש כך - נממש את הממשק *IPolygon* בשניהם, ונוסיף האצלה לכל אחד מהם. נראה כי אם נבקש *timePassed* קורה משהו מוזר - לאן הוא הולך שכותבים *decorated.timePassed*? הרי זה תפקידו לסובב את המשולש. באופן דומה קורה עם *timePassed* במחלקה השנייה. מה קשתן עושה? קודם כל - תעשה מה שלימדו אותך לעשות, אני מבטיח לשמור על ההתנהגות ואז אני ארחיב עליה! כלומר - אם המשולש יודע להסתובב, שיוסתובב. אח"כ נוסיף עליו ונסובב אותו שוב - **אני שומר תכונות מקוריות ורק מרחיב אותן**. או לחלופין - אם יש כבר נקודות בתוך המשולש, אנחנו נצייר אותן כי זה התכונה שלו ואנחנו נרחיב על כך ונוסיף עוד נקודות.



נראה דוגמה -

```
ITrianglerst = newRotatingTriangle(newSpottedTriangle(newEquilateralTriangle(...));
```

מה יצרנו כאן? קודם כל יצרנו משולש שווה צלעות - אח"כ קישתנו אותו והרחבנו אותו בנקודות, ואח"כ קישתנו אותו שוב פעם נוספת והפכנו אותו למשולש שיועד להסתובב! מתחילים מבפנים החוצה.

כמה סוגי משולשים נוכל ליצור? כמה שנרצה! אנסוף! נוכל ליצור איך וכמה שנרצה ונוסיף את הקישוטים עליו מלמעלה - כלומר ניצור *base* ואז נקשט אותו כמה שנרצה! זהו פתרון טוב לכך שלא יכולנו לרשת משתי מחלקות ולכן זהו הפתרון הטוב.

נשים לב שאין לנו הבטחה שייגיע שם בהכרח משולש - יכול להגיע כל *polygon*. איך נמנע ממצב כזה? נגדיר *interface* חדש *ITriangle* שירחיב את *IPolygon* ולא יוסיף לו כלום - עכשיו כל הקשתנים יהיו חייבים לממש *ITriangle*. כל משולש כעת יממש את *ITriangle* ולא את *IPolygon* ולכן מרובעים למשל לא יוכלו להכנס לבפנים. כי הם מסוג *IPolygon*.

בעיה נוספת - יש לנו הרבה שכפול קוד: ניצור *abstract - class* לכל המשותפים. אח"כ המחלקות האחרות ירחיבו אותו וישכתבו מחדש מה שצריך - בהתאם. כך -



# EXAMPLE: TRIANGLES

```
public interface ITriangle extends IPolygon { }
```

```
public abstract class TriangleDecorator implements ITriangle {
    private ITriangle decorated;
    public TriangleDecorator(ITriangle t) {
        this.decorated = t;
    }
    public void timePassed() {
        decorated.timePassed();
    }
    public void drawOn(DrawSurface d) {
        decorated.drawOn(d);
    }
    public double getArea() {
        return decorated.getArea();
    }
    ...
}
```

נדייר TriangleDecorator  
abstract כדי לא לאפשר ליצור  
אובייקטים מסוג זה. המחלקה  
מכילה האצלות עבור כל  
המתודות של ITriangle.

```
public class RotatingTriangle extends TriangleDecorator {
    public RotatingTriangle(ITriangle t) {
        super(t);
    }
    @Override
    public void timePassed() {
        super.timePassed();
        // add rotation
    }
}
```



```
public class SpottedTriangle extends TriangleDecorator {
    public SpottedTriangle(ITriangle t) {
        super(t);
    }
    @Override
    public void drawOn(DrawSurface d) {
        super.drawOn(d);
        // add spots
    }
}
```



## ס יכום: האצלה לעומת הורשה

### ית רונות:

- פתרון גמיש: תומך בשימוש חלקי במתודות ותומך בשימוש בהרבה מחלקות.
- דינמי: הטיפוס המאוצל יכול להבחר בזמן ריצה, בטוח יותר, מפחית את התלות בין הטיפוסים ועמיד לשינויים עתידיים.

### חסרונות:

- איבדנו את  $is - a$  ואת יתרונות הפולימורפיזם
  - מעט פחות יעיל
  - פחות קריא וקשה למעקב
- נשתמש בהורשה כאשר: אנחנו מתכננים היררכיית מחלקות משלנו מתקיימים עקרון  $is - a$  ועקרון ההחלפה מעוניינים שכל תכונות ההורה, גם העתידיות, יחולו על הילדה כאשר אין הור שה חלקית או מרובה ובקיצור, בזהירות!  
אחרת נעדיף האצלה

## 2 דפוס שני: Factory – Design – Pattern

### דוגמה ראשונה:

מדובר בדפוס של מפעל - כלומר בכל פעם ה *pattern* מייצג אובייקטים לפי ביקוש המשתמש. נתבונן בדוגמה של משחק "איקס עיגול". כמו בדוגמה כאן. מכרנו את המשחק ללקוח - הוא חזר ונמא ס לו שהיוזר לא יכול להכניס את הבחירה בעצמו לשיבוץ המקום. *user* משעמם אותי - נרצה *smartUser*. איך נממש? צריך לעדכן את המשחק.

# FACTORY DESIGN PATTERN – TIC-TAC-TOE

```
public class Game {
    private static final int ROWS = 3;
    private static final int COLS = 3;
    private Board board = new Board(ROWS, COLS);
    Player[] players = new Player[2];

    public Game() {
        players[0] = new Player(board);
        players[1] = new Player(board);
    }

    public void play() {
        int maxPlays = ROWS * COLS;
        for (int i=0; i<maxPlays; i++) {
            int currentPlayerNo = i % 2;
            Cell cell = players[currentPlayerNo].play();
            board.add(cell, currentPlayerNo);
            board.print();
            if (board.getWinner() != -1)
                break;
        }
    }

    public int getWinner() {
        return board.getWinner();
    }
}
```

נניח שמחרנו את המשחק שלנו ללקוח, וחלקו מבקש להוסיף שני סוגים של שחקנים – SmartPlayer ו-HumanPlayer באופן חכם, ו- HumanPlayer שמבקש מהמשתמש לבחור תא על הלוח.

ניצטרך לבנות מחלקת Game עבור כל צירוף אפשרי של שחקנים... האם אפשר אחרת?

נממש מחלקות עבור שחקנים

```
public class Player {
    private Board board;
    public Player(Board board) {...}
    public Cell play() {...}
}

public class SmartPlayer {
    private Board board;
    public SmartPlayer(Board board) {...}
    public Cell play() {...}
}

public class HumanPlayer {
    private Board board;
    public HumanPlayer(Board board) {...}
    public Cell play() {...}
}
```

```
public class GameSS {
    SmartPlayer pX, pO;
    public GameSS() {
        pX = new SmartPlayer(board);
        pO = new SmartPlayer(board);
    }
    ...
}

public class GameHH {
    HumanPlayer pX, pO;
    public GameHH() {
        pX = new HumanPlayer(board);
        pO = new HumanPlayer(board);
    }
    ...
}

public class GameHS {
    HumanPlayer pX
    SmartPlayer pO;
    public GameHS() {
        pX = new HumanPlayer(board);
        pO = new SmartPlayer(board);
    }
    ...
}
```



נ וכל ליצור *Iplayer* interface שם יהיו *player*, *smartPlayer*, *HumanPlayer* כדוגמה בצד ימין יראה המחלקה לאחר שכתוב. זה לא טוב! בבנאי של *Game* אנחנו חייבים להגדיר מראש שני אובייקטים של *players*. כאן בדוגמה אתחלנו זאת מראש - מי יודע מה ה-*user* רוצה ואיך הוא רוצה שהשחקנים במשחק יהיו? נקבל מה-*user* ב-*args* מה הוא רוצה שיהיה. ניצור ממשק חדש - *supplier* - אותו יממשו שלוש מחלקות כמפורטות מטה, אח"כ ב-*main* כשניצור נוכל לקבל שני *supplier* ובעזרת המתודה *get* נתחייב כי אכן יהיה קל להוסיף כמה סוגי שחקנים שנרצה ללא ש ינויים דרמטיים בקוד.



# FACTORY DESIGN PATTERN – THE

בהמשך הקורס נלמד  
כלים שאפשרו לנו  
לשכלל את הפתרון הזה

פתרון כזה נקרא  
**Factory Design Pattern** - מאפשר  
יצירת אובייקט מבלי לדעת מהו  
הטיפוס הספציפי של האובייקט.  
זה פתרון לפי עקרונות  
של OOP.



```
public class Game {
    private static final int ROWS = 3;
    private static final int COLS = 3;
    private Board board = new Board(ROWS, COLS);
    IPlayer[] players = new IPlayer[2];

    public Game(Supplier s1, Supplier s2) {
        players[0] = s1.get(board);
        players[1] = s2.get(board);
    }
    ...
}
```

```
public interface Supplier { IPlayer get(Board board); }

class PlayerSupplier implements Supplier {
    public IPlayer get (Board board) { return new Player(board); }
}

class SmartPlayerSupplier implements Supplier {
    public IPlayer get (Board board) { return new SmartPlayer(board); }
}

class HumanPlayerSupplier implements Supplier {
    public IPlayer get (Board board) { return new HumanPlayer(board); }
}
```

```
public static void main(String [] args) {
    switch(args[0]) {
        case "SS":
            Game game = new Game(new SmartPlayerSupplier(), new SmartPlayerSupplier());
            break;
        ...
    }
    game.play();
    if(game.getWinner() < 0) System.out.println("Tie");
    else System.out.println("Winner is player " + winner);
}
```

כתיבה כזו עם *supplier* היא בדיוק *Factory – design – pattern*! כשנרצה לפעול כך נשתמש בשיטה זו.

## דפוס *observer*:

**דוגמה:** לכל אחד מאיתנו יש חשבון אינסטגרם עם עוקבים, כל אחד מהעוקבים הוא *observer*. כל אחד שעוקב אחרינו באינסטגרם צריך להיות מסוגל להצטרף (לעקוב) ולהפוך ל-*observer* או כאשר הוא רוצה להוריד עוקב - שיוכל להוריד מאיתנו עוקב וכן כאשר אנחנו מפרסמים *posts* נרצה שכל אחד מהעוקבים יקבל הודעה על כך! נרצה שהעוקבים "ינהלו" את עצמם. במצב כזה נשתמש בדפוס הנ"ל.

**דוגמה נוספת:** נרצה לשמור סטטיסטיקות על משחק - מס' נצחונות לכל שחקן ורצף נצחונות מקסימלי. איך נממש זאת? ה-*observer* שלנו כאן יהיה *StatisticsTracker*. במחלקה *Game* נוסיף *list* של *listeners*, כמו כן נממש שם מתודות של הוספת ומחיקת *listener*. בכל רגע שקורה מאורע נקרא לכל ה-*listeners*.

# OBSERVER DESIGN PATTERN

```
public class StatisticsTracker implements GameEventListener {
    private int numTurns = 0;
    private int longestWinSequence = 0;
    private int[] numWins = new int[2];

    public void turnStarted() {
        numTurns++;
    }
    ...
    public void turnEnded(int turnWinner) {
        numWins[turnWinner]++;
    }
    ...
}
```

```
public interface GameEventListener {
    void turnStarted();
    void turnEnded(int turnWinner);
}
```

```
public abstract class Game {
    private Player p1, p2;
    private List<GameEventListener> listeners;

    public Game(String s1, String s2) {
        this.p1 = new Player(s1);
        this.p2 = new Player(s2);
        listeners = new LinkedList<>();
    }

    public Player playOneTurn(Player p1, Player p2) {
        notifyTurnStarted();
        ...
        notifyTurnEnded(turnWinner);
        return turnWinner;
    }

    public void addListener(GameEventListener lst) {
        listeners.add(lst);
    }
    public void removeListener(GameEventListener lst) {
        listeners.remove(lst);
    }
    private void notifyTurnStarted() {
        for (GameEventListener l : listeners)
            l.turnStarted();
    }
    private void notifyTurnEnded(int turnWinner) {
        for (GameEventListener l : listeners)
            l.turnEnded(turnWinner);
    }
}
```

נוצר רשימת מקשיבים  
שנמצא יחד עם הליקס  
כל מי שמעוניין לקבל  
התקשרות

ברגע שיש מקשיב,  
מיוזם את כל ה-  
listeners על המודל הזה

- העצמים המודווחים – observables
- העצמים המקשיבים לשינויים ומעבירים אותם – observers
- המודווחים לא צריכים לדעת מי מקשיב
- מקשיבים יכולים להתווסף או להיטול
- המקשיבים יכולים להקשיב להרבה מודווחים

המיוצר שומע על התקשרות  
GameNarrator,  
EmailDigestCreator וכן  
GameEventListener

```
public static void main(String[] args) {
    GameNarrator narrator = new GameNarrator();
    StatisticsTracker stats = new StatisticsTracker();
    EmailDigestCreator email =
        new EmailDigestCreator("John", "john@gmail.com");

    Game game = new Game();

    game.addListener(narrator);
    game.addListener(stats);
    game.addListener(email);

    System.out.println("The winner: " + game.play());
}
```

מסירה את כל  
המקשיבים למחזור

## דפוס *iterator*:

\* יש לנו אוסף של אובייקטים ונרצה לעבור עליהם אחד אחרי השני. נרצה לבצע על כל אחד מהם משהו. מה אם נשתמש ב-*ArrayList*? סבבה. אבל מה אם נעשה זאת על *LinkedList*? זה לא יעיל, כיוון שאנחנו ניגשים ב- $O(n)$  לכל איבר לחפש אותו (הרי המחשב מחפש אותו כי הוא לא רציף בזכרון גם אם נעשה *get(i)* וזה קורה עבור  $n$  איברים וסה"כ זה  $O(n^2)$ . עוד יותר חמור - לא בכל מבנה נתונים הם אחד אחרי השני - למשל טבלת האש.

\* נרצה אובייקט בשם *iterator* שידע לעבור בצורה יעילה על מבנה הנתונים (מבלי להכיר מראש את מבנה הנתונים שכן נרצה לשמור על *encapsulation*!).

\* באופן תאורטי יכולנו לחשוף את המבנה הפנימי של המידע שלנו - לשבור את ה-*encapsulation* ואכן לבצע יעיל. אבל זה לא טוב! גוגל חושפת את מבני הנתונים בהם היא משתמשת? ממש לא.

\* לכל מבנה נתונים נבנה *iterator* שיוכל לעבור על המבנה (תמיד יראה אותו דבר!) כך שנוכל בכל מבנה נתונים להשתמש בו.

יש ממשק שכתבו לנו מראש וניתן להשתמש בו -

```
public interface Iterator<E>{
    boolean hasNext();
    E next();
}
```

כמו כן, *hasNext* תחזיר האם יש איבר הבא אחרי, וכן *next* תחזיר את האובייקט מהסוג הגנרי הבא אחרי במבנה הנתונים. נשים לב שנוכל להוסיף לממשק באופן תאורטי גם *remove* אך לא חובה לממש אותה כמובן.

את אותו *interface* נצטרך לממש במבנה הנתונים שלנו!

נראה דוגמה למימוש אותה מחלקה עם *ArrayList*:

```

1 class ArrayListIterator<E> implements Iterator<E> {
2
3     private ArrayList<E> list;
4     private int currentIndex;
5
6     public ArrayListIterator(ArrayList<E> list){
7         this.list = list;
8         this.currentIndex = 0;
9     }
10
11     public boolean hasNext() {
12         return (currentIndex < list.size());
13     }
14
15     public E next() {
16         E result = this.list.get(this.currentIndex);
17         this.currentIndex += 1;
18         return result;
19     }
20 }

```

איך נשתמש בזה כעת?

```

ArrayList<Integer> list = new ArrayList<Integer>();
addRandomValues ( list ); // add random values to our list
int sum = 0;
Iterator<Integer> itor = new ArrayListIterator<Integer>( list );
while ( itor . hasNext () ){
    Integer value = itor . next () ;
    sum += value ;
}

```

קטע הקוד הזה יהיה נכון לכל מבנה נתונים כעת!!!, גם שמרנו על עקרון ההכמסה וגם קל מאוד להחליף מבני נתונים.

\*ניתן להשתמש ב-*itertor* גם עבור דברים כמו חישוב האיבר ה- $n$  של סדרת פיבונאצ', נגדיר בקונסטרקטור שנשלח איבר כלשהו  $n_i$  ונחזיר איבר  $n_i$  בפיבונאצ'. איך נממש? *hasNext* זה כל עוד יש לי איברים ח דשים בפיבונאצ' לייצר (הרי בשביל לחשב איבר אני צריך לייצר את  $n-1$  קודמיו. נראה גם שנוכל לממש די בקלות את *next* בהינתן  $a = F(n-1)$  וכן  $b = f(n-2)$  נראה כי

```

int current = a
int sum = a + b
a = b
b = sum
remaining-
return current

```

וכך נוכל לחשב בקלות את פיבונאצ' בעזרת ה-*itertor*, נעבור על כל סדרת פיבונאצ' עד מס'  $n_i$  שנבקש. ניתן לחשב כל סוגי סדרות כך! ושוב - קטע הקוד בסוף של ה-*itertor* מאוד זהה!!! זה יתרון עבורנו.

*Nested – class*

יש עדיין ב-*itertor* בעיה. לפעמים הוא כן צריך לדעת את המבנה הפנימי לפחות עבור ה-*itertor*. כלומר שרק הוא ידע, אבל כרגע אי אפשר לחשוף אותו כי איננו חלק מהמחלקה. כאן נשתמש ב-*itertor* בלי לשמוע על הכמסה. מדובר על מחלקה בתוך מחלקה!!!

```

1 class OuterClass {
2
3     void func(){
4         new InnerClass();
5     }
6
7     private class InnerClass {
8         ...
9     }
10 }

```

מה חדש כאן? המחלקה הפנימית כן מכירה את התכונות *private* של המחלקה הראשית! אם נגדירה כמחלקה פרייבט גם לא נוכל ליצור אובייקט ממנה. כעת נוכל ליצור את ה-*iterator* בפנים שיכיר את המבנה ולא ישבר עקרון ה-*encapsulation*.  
 כן היינו רוצים לקרוא ל-*iterator* באופן כללי מבחוץ - ולכן אם נגדיר את המחלקה הפנימית כ-*static*, זו מחלקה שמוגדרת פר המחלקה כולה המקורית שמכילה אותה.  
**\*\*\*הערה - יש ממשק נוסף בשם *Iterable* שהוא אומר שניתן להשתמש ב-*iterator* על מבנה הנתונים.**

**\*\*\*** טוב לדעת - הסיבה שמותר לעשות *for - each* כמו *for(Integer - num : collection)* היא שמאחורי הקלעים ממומש *iterator* של המחלקה שנקרא, וכמו כן מותר לעשות *for - each* רק על אובייקטים שהם *iterable*.

## הרצאה 8 - *Cloning&Immutability*:

*Immutable* - לא ניתן לשינוי  
*Mmutable* - כן ניתן לשינוי.

ישנם שפות רבות שהינם *Immutable*, כלומר הגדרנו  $x = 3$ , זהו - אי אפשר לשנות אותו. נגלה עליהם בהמשך וגם מדוע צריך את זה ומדוע לא אפשרי לשנות. נגלה על כך בקורס "שפות תכנות".  
**\*Clone**: שכפול. יש לי אובייקט? נרצה להכפיל אותו ולקבל אחד זהה כמוהו.  
**נתבונן בדוגמה שגויה -**

```

public class Exam { private List<Student> students = new List<>(); private Lecturer lecturer;
public void addStudent(Student s) { students.add(s); }
public List<Student> getAllStudents() { return students; }
public Lecturer getLecturer() { return lecturer; } }

```

מה הבעיה בקוד? נראה שלא נשמר עקרון ה-*encapsulation*! אנחנו מחזירים את המורה או הסטודנטים ומישהו יכול לגשת למידע הזה ולשנות אותו כי שלחנו *refrence*!  
**איך נפתור את הבעיה?** נרצה ליצור העתק של הסטודנט על מנת שמישהו מבחוץ לא יוכל לגשת, אבל זה הרבה זכרון שנתפס וכן אנחנו לא יודעים מה הטיפוס המדויק של הסטודנט. כאשר ניצור העתק לא נדע האם זה סטודנט למדעי המחשב או מתמטיקה... לא עובד!

**ב- *Object* - class מוגדרת מתודת *clone***  
 המתודה הינה *Protected*  
 מחזירה *Object*

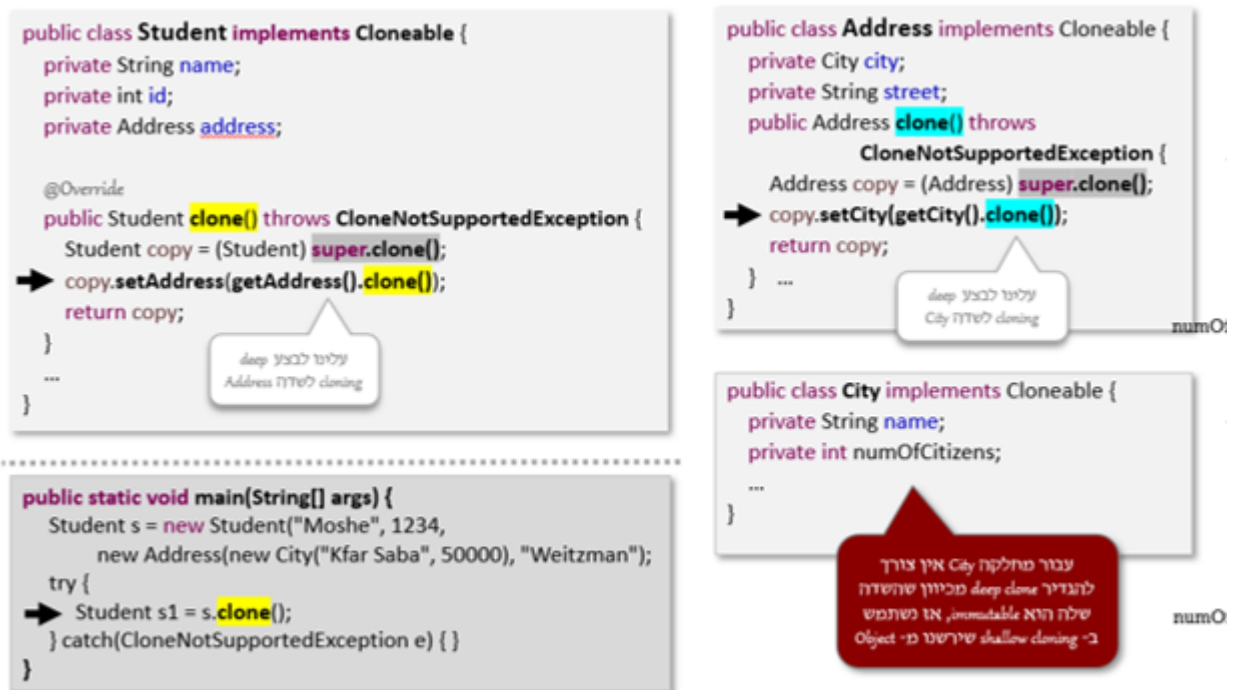
יוצרת עותק שטחי (*shallow - copy*) של כל השדות של האובייקט המועתק  
 המתודה זורקת חריג עבור מחלקה שאינה מממשת *Cloneable - interface* אם אובייקט דורש העתקה עמוקה (*deepcopy*), יש לדרוס אותה  
*Cloneable* - ממשיך ריק, מסמן מחלקות שתומכות ב-*cloning*  
 זו מתודה ג'אוית שקיימת בתוך *Object*, החתימה שלה היא כך-

*protected - Object - clone() - throws - CloneNotSupportedException.....*

על מנת לאפשר לclass לגשת לclone הוא חייב לממש את הממשק cloneable - זהו ממשק ריק! לא מתחייבים לכלום, אנחנו רק מקבלים אישור מהjvm. הexception יזרק כאשר ננסה להפעיל את המתודה על מחלקות שלא ממשות את cloneable.

**נחזור לדוגמה ממקודם - כעת אם ניצור סטודנט כלשהו בmain ואז נגדיר s2 = student - s1.clone();** נקבל שנפעיל את הclone של Object ויווצר העתקה - זה נקרא shallow - copy, העתקה רדודה! זה לוקח את כל השדות ופשוט מעתיק את הערכים asIs בלי לחשוב על ההשלכות. בואו נחשוב על ההשלכות... יש לסטודנט שם, משה. אנחנו יודעים כבר שאם יש לנו שתי מחרוזות באותו שם הן נשמרות בזכרון באותו מקום! כלומר קיבלנו שלכל סטודנט יש id נפרד, אך בשם הם מצביעים אל אותו השם (כי ככה עובד string!). קיבלנו אובייקט וחצי!!..... במקרה שלנו זה תקין! כיוון שstring הוא immutable ולא ניתן לשנותו. מסקנה: אם יש לנו שדות פרמיטיביים שהם immutable זה בסדר לנו. אם אחד ינסה לשנות את שמו הוא יקבל refrence חדש לשם החדש.

**איך נגדיר Clone עבור Class - Student?** כעת הוספנו אובייקט בשם כתובת לשדות, נתחיל בshallow - clone רגיל ואח"כ נצטרך לעשות שוב clone לאובייקט השני - כתובת, זה נקרא deep - clone. יחד עם זאת נצטרך לדרוס את המתודה אצל address. מדוע? גם שם יש אובייקט City! נצטרך לעשות שוב shallow - clone, לקרוא לclone עם super! ואח"כ לבצע clone עבור City. ואז סיימנו - אין צורך לעשות clone בCity כי השדה שלה הוא immutable. כעת כשנרצה ליצור סטודנט בmain, נוכל להשתמש במתודה clone כמו בדוגמה על מנת להעתיקו. (הדוגמה עליה דיברתי מצורפת כאן בתמונה)



לסיכום - תמיד נדרוס את הclone של object עד שנגיע לאובייקט שהוא immutable - ושם לא נצטרך לעשות deep - Clone.

כעת, נוכל לפתור את הבעיה שאיתה פתחנו את הנושא:

```
public void addStudent(Student s) {
    students.add((Student)s.clone());
}
public List<Student> getAllStudents() {
    List<Student> copy = new List<>();
    for(s : students)
```



```

copy.add((Student)s.clone());
return copy; }
public Lecturer getLecturer() {
return (Lecturer)lecturer.clone(); } ... }

```

באמצעות *clone* אבל שכפול הרשימה לוקח הרבה זמן - והרבה זכרון! אולי נמצא לזה פתרון אחר?.....

## Immutability

פתרון חלופי לבעיה הקודמת. בואו נניח שהסטודנט היה *immutable*, זה היה הרבה יותר קל עם *clone* אחד. אם המורה, גם כן "ל..."

יצרנו מחלקה שהיא *ImmutableList* - וכן במחלקה *exam* יצרנו את הרשימה כאובייקט של המחלקה *ImmutableList*. כלומר - רשימה שאינה ניתנת לשינוי. גם היא תצטרך לממש את *cloneable*. השדה שבפנים יהיה רשימה שהיא *immutable* אבל היא תהיה *private* ואז לא ניתן יהיה לשנות מבחוץ ואם נבנה את המחלקה בצורה נכונה נשמור על *encapsulation*. שוב, נהיה חייבים לממש גטרים וסטרים וכן *add* ו*remove*.

**\*הערה** - במתודה *add* יש שורה *copy.l.add* - מה קורה כאן? אנחנו מבצעים את השינוי על הרשימה *l* של *copy*!

# IMMUTABILITY

```

public class Student implements Cloneable{
    private String name;
    private int id;
    ...
}

```



```

public class CSStudent extends Student {...}
public class BiologyStudent extends Student {...}
public class MathStudent extends Student {...}

```

```

public class Lecturer implements Cloneable{
    private String name;
    private int id;
    ...
}

```



```

public class Exam {
    private ImmutableList<Student> students = new ImmutableList<Student>();
    private Lecturer lecturer;
    public void addStudent(Student s) { students.add(s); }
    public ImmutableList<Student> getAllStudents() { return students; }
    public Lecturer getLecturer() { return (Lecturer)lecturer.clone(); }
}

```

```

public class ImmutableList<T> implements Cloneable{
    private List<T> l;

    public ImmutableList() { this.l = new LinkedList<>(); }
    public ImmutableList(List<T> l) { this.l = (List<T>)l.clone(); }
    public ImmutableList<T> add(T item) {
        ImmutableList<T> copy = (ImmutableList<T>)this.clone();
        copy.l.add((T)item.clone());
        return copy;
    }

    public T get(index i) { return (T)l.get(i).clone(); }

    @Override
    public Object clone() throws CloneNotSupportedException{
        ImmutableList<T> copy = new ImmutableList<>();
        for(item : l)
            copy.l.add((T)item.clone());
        return copy;
    }
    ...
}

```

אנחנו מוסיפים item  
ול שותק שלנו, ולא  
לעצמינו

כעת נזכר, מדוע בכלל בנינו את המחלקה? להמנע מבזבוז זמן העתקות - וכאן בדיוק אנחנו

מבצעים אנסוף העתקות! כל *add* אנחנו מעתיקים את כל הרשימה.... חבל על הזמן ועל הזכרון. ולכן נפצל למקרים - זה תלוי במקרה ובאפליקציה שלנו! **שני הפתרונות שלנו גרועים**. מה פחות גרוע? אם אני פחות מעדכן את הרשימה אז נשתמש בש יטה הזו ואם אנחנו הרבה מעדכנים את הרשימה נשתמש בפתרון הקודם.

\* מומלץ מאוד (לא חובה ולא עקרונות *oop*) במחלקה *ImmutableList* להגדיר כ *final* את המחלקה (לא ניתן להרחיב אותה - כי ההרחבה ע לולה להרוס את תכונת *immutability*) וכן את כל השדות כ *final* - ואם ננסה לשנות את ה *reference* שלי נקבל שגיאה (זה הרי היה בטעות) - ניגש לשיטה זו רק כאשר לא נרצה כמעט אם בכלל לעדכן את הרשימה.

## Immutability—disadvantages

1. **חסרון ראשון** - בדוגמה כאן הסיבוכיות תהיה  $O(n^2)$  למרות שיש רק  $n$  איטרציות, מדוע? בכל איטרציה אנחנו משכפלים מחדש את המחרוזת של נו ולכן זה עולה  $O(n)$  כפול  $n$  איטרציות.

```
String s = "";
for (int i = 0; i < n; i++)
    s = s.concat("a");
```

```
concat(s, "")
s -> "a"
concat(s, "a")
s -> "aa"
concat(s, "aa")
s -> "aaa"
concat(s, "aaa")
s -> "aaaa"
concat(s, "aaaa")
s -> "aaaaa"
...
```

Runtime of this code is  $O(n^2)$  since each call to `concat()` results with the new string

Let's see how Java fixed this problem for class String

**פתרון:** נפעיל שיקול דעת, במקרים כמו כאן אנחנו נרצה את ה *string* להיות *mutable*! לשם כך ג'אווה מודעת ויצרה *StringBuilder* למקרים כאלו ואם נריץ את הקוד הבא:

```
StringBuilder builder = new StringBuilder(); // define empty string
for (int i = 0; i < n; i++) builder.append("a ");
String s = builder.toString(); //
```

נקבל שהסיבוכיות אכן  $O(n)$ .

2. *Immutable - Stack*: ללא העתקות! נעבוד הרבה יותר קשה ויצירתי אך יש קבוצת אנשים שחשבו והשקיעו בזה ויצרו מבני נתונים *Immutable* שהוא גם יעיל:

# IMMUTABLE STACK

```
public final class EmptyStack
    implements ImmutableStack {

    public EmptyStack() {}
    public ImmutableStack push(int data) {
        return new ImmutableStackClass(this, data);
    }
    public ImmutableStack pop() {
        throw new IllegalStateException("pop empty!");
    }
    public int top() {
        throw new IllegalStateException("top empty!");
    }
    public int size() { return 0; }
    public boolean isEmpty() { return true; }
}

public static void main(String[] args) {
    ImmutableStack s1 = new EmptyStack();
    s1 = s1.push(10).push(20);
    ImmutableStack s2 = new EmptyStack();
    s2 = s2.push(30).push(40);
}

// Let's create several stacks

public final class ImmutableStackClass
    implements ImmutableStack {

    private final ImmutableStack prev;
    private final int data;
    private final int size;

    public ImmutableStackClass(ImmutableStack prev,
                               int data) {

        this.prev = prev;
        this.data = data;
        this.size = prev.size()+1;
    }

    public ImmutableStack push(int data) {
        return new ImmutableStackClass(this, data);
    }

    public ImmutableStack pop() { return prev; }
    public int top() { return data; }
    public int size() { return size; }
    public boolean isEmpty() { return false; }
}
```

**במימוש כאן כל הפעולות ב  $O(1)$ .** מה קורה כאן בעצם? המחסניות נבנות אחת על השנייה כך שהמחסניות הריקות נמצאות למטה. מיהו *prev*? המחסנית הקודמת. בכל מחסנית יש "איבר אחד" פלוס האיברים שנמצאים מתחתיה. זה בדיוק רשימה מקושרת... כל אחד זוכר את זה שלפניו עד המחסנית הריקה. מחסנית שמיוצגת ע"י רשימה מקושרת. מה קורה במחיקה? אנחנו רק נחזיר את הכתובת של הקודם, וה *garbage – collector* ינקח אותו כי אף אחד לא מצביע עליו! (מומלץ לראות את ההקלטה של מרינה לנושא זה!)

כל השדות הינם *private* השדות נוצרים בתוך המחלקה ולא יוצאים החוצה איברי המחסנית הינם מטיפוס מקובע *Integer* מתודות שמשנות את התוכן של המחסנית הינן פרטיות נקראות אך ורק כחלק מיצירת עצם או העתק שלו

**תכונת Singleton:** הגבלת מס' המופעים של המחלקה למופע יחיד (ניתן גם למס' קבוע של מופעים) מדוע זה חשוב? בדוגמה שראינו למעלה נרצה להגביל את מס' הפעמים שאפשר ליצור מחלקה ריקה - ולכן נשתמש בתכונה זו. מופע זה ישמר כשדה סטטי (יחיד) של המחלקה ויוחזר ע"י מתודה סטטית של המחלקה. כלומר נגדיר את הבנאי להיות *private*! וניצור מתודה *create*. למשל היצירה תראה כך -

*private – final – static – empty = new – NameClass();*

ואז נממש מתודה שתהיה אחראית ליצור את המופע היחיד הזה

*Public – static – create(){return – nameClass.empty;*



## סיכום של תבניות עיצוב

### 1. *Observer/listner*:

מוטיבציה - נרצה אובייקט שנוכל "להסתכל" עליו או "להאזין לו". כלומר נרצה אובייקט שכאשר נשנה משהו, כל המקומות שקוראים לו ישנו גם כן משהו. יש קשר בינו לבין רבים. בכל פעם שיתבצע בו שינוי הוא יבצע *notify* לכל האחרים שיתעדכנו גם כן. למשל, כאשר כדור יעבור מהירות מסויימת נבצע *notify* שיעדכן.

למעשה התבנית מאפשרת לאובייקט אחד להודיע לאובייקטים אחרים (הצופים בו) על שינויים במצבו. למעשה כך נוצר קשר דינמי בין האובייקט לבין הצופים. הצופים יקבלו עדכונים באופן אוטומטי ברגע בו הם מתרחשים.

**האובייקט הנצפה** - *subject*: מנהל רשימה של *Observers*. מאפשר ל-*Observers* להירשם או להסיר את עצמם מהרשימה. מכיל פונקציה *notify()* ששולחת עדכון לכל ה-*Observers* הרשומים. **הצופים** *Observers*: אובייקטים שמחוברים ל-*Subject*. מקבלים הודעה אוטומטית כאשר יש שינוי ב-*Subject*. מיישמים ממשק מסוים, לדוגמה, *update()*.

### 2. *Decorator* (האצלה):

מדובר בקשתן. נשתמש כאשר נרצה "לקשט" את האובייקט בתכונות נוספות. למשל נרצה להוסיף צבע לכדור, או לסובב אותו וכדומה. נוסיף מחלקות שיקשטו אותו מסביב. נובע מכך שלא ניתן לרשת משני מחלקות ולכן הפתרון הוא להשתמש בתבנית עיצוב - נוסיף שדה של *ball*. הקשתן מאפשר ליצור מחלקת בסיס עם תכונות והתנהגויות שיוגדרו מראש, ולהוסיף תכונות רצויות (למשל לצבוע אותו) באמצעות הקשתן. בנראות הקוד זה נראה כאילו ממש אנחנו מקשטים את האובייקט הפנימי ביותר החוצה. הקשתן מרחיב את ההתנהגות בזמן הריצה, לא בזמן קומפילציה, לכן יותר גמיש ודינמי.

### 3. *Factory*:

מדובר בדפוס של מפעל - כלומר בכל פעם ה-*pattern* מייצג אובייקטים לפי ביקוש המשתמש. נפריד בין יצירת המופע לבין שימוש במופע. איך הפקטורי יהיה בנוי? יש ממשק או מחלקת אב אבסטרקטית *factory* עם מתודה *create* ומחלקות שיממשו אותה / יורשות אותה ומהוות מפעלים של אובייקטים ספציפיים.

מתי נשתמש בתבנית זו? כאשר מתודה במחלקה מסויימת מבקשת ליצור מופע של אובייקט ספציפי, שאיננו יודעים מראש מיהו, אלא זה תלוי בבחירת המשתמש (למשל בתוך משחק איקס עיגול) ולכן נשתמש במפעל. המתודה תקבל בתוך פרמטר את המפעל המתאים ותיצור את האובייקט כך:

```
shape - s = factory.create();
```

בתבנית עיצוב זו יש חלוקת תפקידים ברורה. המפעל אחראי על יצירת המופע, והמתודות אחראות על שימוש בו.

נשתמש באיזשהו *supplier* שיעזור לנו לחלק את האובייקט. למשל אם אנחנו בתוך משחק והמשתמש מבקש אובייקט משולש צהוב ולא משולש אדום, ה-*supplier* יהיה אחראי לתת לו אובייקט זה. כך המפעל הזה אחראי על היצירה של המופע, בזמן ששאר המתודות נכתבות אותו דבר ללא שכפול קוד ויודעות שיקבלו אובייקט כלשהו.

### 5. *Singleton*:

הגבלת מספר המופעים של אובייקט למופע יחיד (או כמה, מס' קבוע). איך עושים זאת? נגדיר במחלקה שדה סטטי בשם *instance* בו את המופע של המחלקה. נהפוך את בנאי המחלקה לפרטי, כך נמנע יצירת אובייקטים מחוץ למחלקה, ניצור מתודה סטטית שתחזיר את המופע, היא תבדוק אם *instance = null*, אם כן היא תקרא לבנאי ותחזיר את המופע למשתמש. אחרת, הוא כבר קיים והיא תחזיר את השדה הסטטי (המופע).

### 6. *Comparator*:

נרצה להשוות בין אובייקטים. התבנית *comprable* (ממשק בג'אוה) משמש לסידור של אובייקטים לפי תבחין מסוים. מחלקות שימשו את המחלקה יצטרכו לממש את המתודה *compareTo* שמבצעת השוואה מסוג מסויים בין האובייקט הנוכחי לאובייקט אחר מאותו הסוג. התבנית השנייה היא אינטרפייס של *compertor* שמשמשת לסידור מותאם אישית של אובייקטים. הממשק מכיל את המתודה *compare* אותה יש לממש שמשווה בין שני אובייקטים מאותו סוג. נרצה ליצור מס' מחלקות כאלו על מנת למיין אובייקטים בדרכים שונות. (מזכיר את הרעיון הגנרי במבוא של מיון)

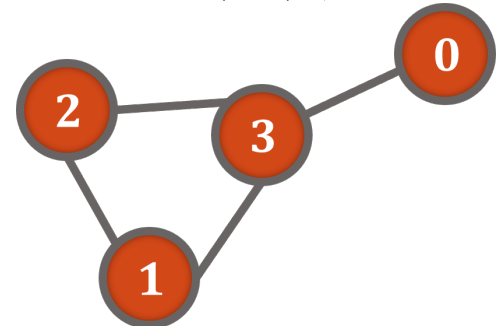
**7. *Iterator*:**

נשתמש בו כאשר נרצה לעבור על אוסף כלשהו מבלי שאנו יודעים את מבנה הנתונים שהאוסף מאוחסן בו.

## הרצאה 9: גרפים

הערה: לא יהיה כאן חומר חדש, גרפים הם שימושיים בחוץ בתעשייה ובכל העולם של מדעי המחשב. נרצה לראות בהרצאה זו כיצד מתכנתים ב*oop* עם גרפים.

**תזכורת:** גרף  $G = (V, E)$  מורכב מקבוצת קודקודים וקבוצת קשתות  $E \subseteq V \times V$ . למשל -



קיים *interface* כבר בג'אוה:

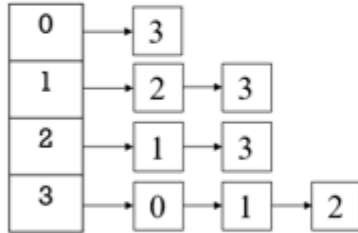
```
public interface Graph {
    public int numberOfVertices();
    public int numberOfEdges();
    ; public boolean containsEdge(int i, int j);
    public void addEdge(int i, int j);
    public void removeEdge(int i, int j);
    public int degree(int v);
    public Set<Integer> neighborsOf(int v);
    public Set<Edge> edgeSet(); }
```

**דרך למימוש גרף:** אפשרות אחת היא כמו במבוא עם רשימה מקושרת, דרך אחרת היא באמצעות מטריצה כמו שמופיע כאן מימין. אם יש מפגש של שתי הקודקודים נסמן זאת עם  $T$  ואחרת אין כלומר  $F$ . למען הפשטות נעבוד עם מחלקה *Edge* שתייצג צלע (קשת), כאשר יש לה *left* ו *right*.



$V = \{0,1,2,3\}$   
 $E = \{(0,3), (1,2), (1,3), (2,3)\}$

We would implement a class for each representation



	0	1	2	3
0	false	false	false	true
1	false	false	true	true
2	false	true	false	true
3	true	true	true	false

public class **GraphAsAdjacencyList** implements Graph {...}

public class **GraphAsAdjacencyMatrix** implements Graph {...}

את שתי השיטות ניתן לייצג באמצעות המחלקות הבאות:

```
public class GraphAsAdjacencyMatrix implements Graph {
    final private int nVertices;
    private boolean[ ][ ] edges;

    public GraphAsAdjacencyMatrix(int n) {
        nVertices = n;
        edges = new boolean[nVertices][nVertices];
        for (int i = 0; i < edges.length; i++)
            for (int j = 0; j < edges[i].length; j++)
                edges[i][j] = false;
    }

    public int numberOfVertices() {
        return nVertices;
    }

    public int numberOfEdges() {
        return edgeSet().size();
    }
    ...
}
```

false	false	false	true
false	false	true	true
false	true	false	true
true	true	true	false

Adjacency Matrix

```
public class GraphAsAdjacencyList implements Graph {
    final private int nVertices;
    private List<Integer> [ ] adj;

    public GraphAsAdjacencyList(int n) {
        nVertices = n;
        adj = new List<Integer>[n];
        for (int i=0; i<nVertices; i=i+1)
            adj[i] = null;
    }

    public int numberOfVertices() {
        return nVertices;
    }

    public int numberOfEdges() {
        return edgeSet().size();
    }
    ...
}
```



Adjacency List

The classes have methods with same implementation. How should we change our design?

Let's not parent d

**מוסכמה בקורס:** כאשר ניצור גרף ניצור גרף שכמות הקודקודים שלו לא ניתנת לשינוי, כלומר - יצרנו גרף עם 100 קודקודים זה יהיה גודלו עד סוף חיי התוכנית.

**\*הערה:** על הנייר היה נראה טוב לייצג גרף עם טבלת גיבוב כך שחיפוש הוצאה ומחיקה יהיה  $O(1)$ , אבל איך נממש פעולה של מה הדרגה של קודקוד? לא יהיה ניתן לדעת כמה שכנים יש לקודקוד. ניתן לעבור על כל הקודקודים כל פעם ולדעת לעשות פעולה זו ב  $O(degree) < O(n)$ . אנחנו לא נשתמש בטבלת גיבוב כי זה לא מתאים לגרפים.

נרחיב את הדיון וניצור קלאס אבסטרקטי ובכל גרף מחדש אנחנו נממש את שלושת המתודות שמטה: **הוספת קשת, מחיקת קשת, וחיפוש קשת.**

# AbstractGraph IMPLEMENTATION

```
public abstract class AbstractGraph implements Graph {

    final private int nVertices;
    public AbstractGraph(int n) { nVertices = n; }

    public int numberOfVertices() {...}
    public int numberOfEdges() {...}
    public int degree(int v) {...}
    public Set<Edge> edgeSet() {...}
    public Set<Integer> neighborsOf(int v) {...}
    // returns true iff i and j are vertices
    protected boolean rangeCheck(int i, int j) {...}
    // returns true iff i is a vertex
    protected boolean rangeCheck(int i) {...}
    public String toString() {...}
    public boolean equals(Object other) {...}
    public Object clone(Object other) {...}

    public abstract boolean containsEdge(int i, int j);
    public abstract void addEdge(int i, int j);
    public abstract void removeEdge(int i, int j);
}
```

HashSet<T> is a Java class

```
public Set<Edge> edgeSet() {
    Set<Edge> edges = new HashSet<>();
    for(int i=0; i<numberOfVertices(); i++)
        for(int j = i+1; j < numberOfVertices(); j++)
            if(containsEdge(i, j))
                edges.add(new Edge(i,j));
    return edges;
}
```

המחלקה *GraphAlgorithm*:

נשתמש כאשר נרצה להריץ אלגוריתמים כלשהם על גרף. כך נממשה:

```
public abstract class GraphAlgorithm {
    protected Graph g;
    public GraphAlgorithm(Graph g) { this.g = g; }
    public abstract Object run();
}
```

דוגמה - כאשר נרצה להריץ אלגוריתם שבודק אם כל דרגות הגרף זוגיות.

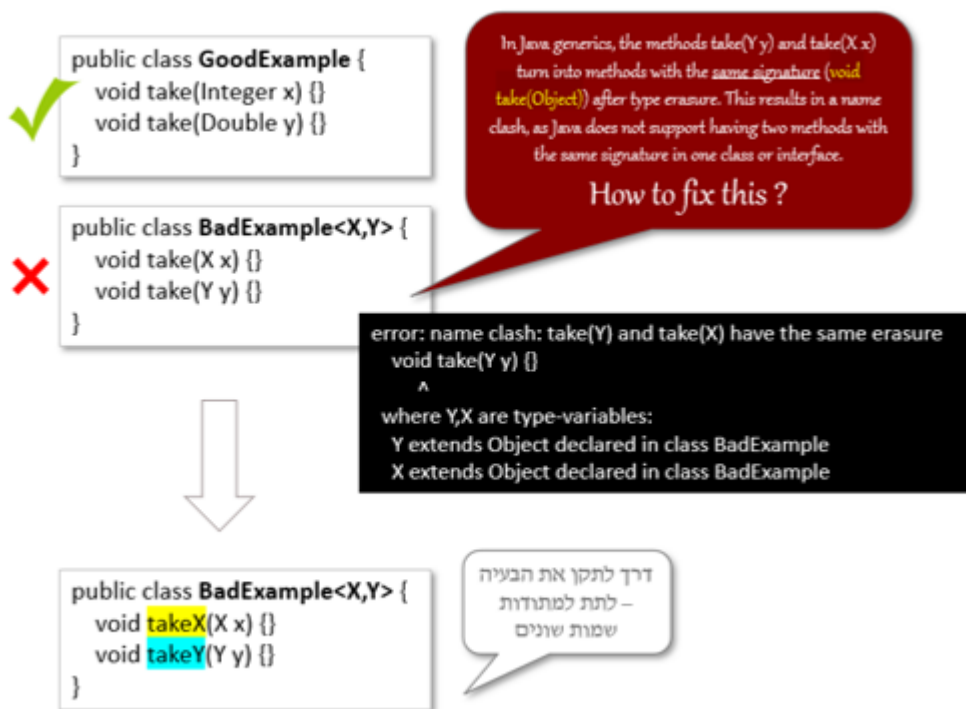
```
public class EvenDegreesAlgorithm extends GraphAlgorithm
```

```
public EvenDegreesAlgorithm(Graph g) { super(g); }
public Object run() {
    for(int v = 0; v < g.numberOfVertices(); v++)
        if(g.degree(v) % 2 != 0)
            return false;
    return true;
}
```

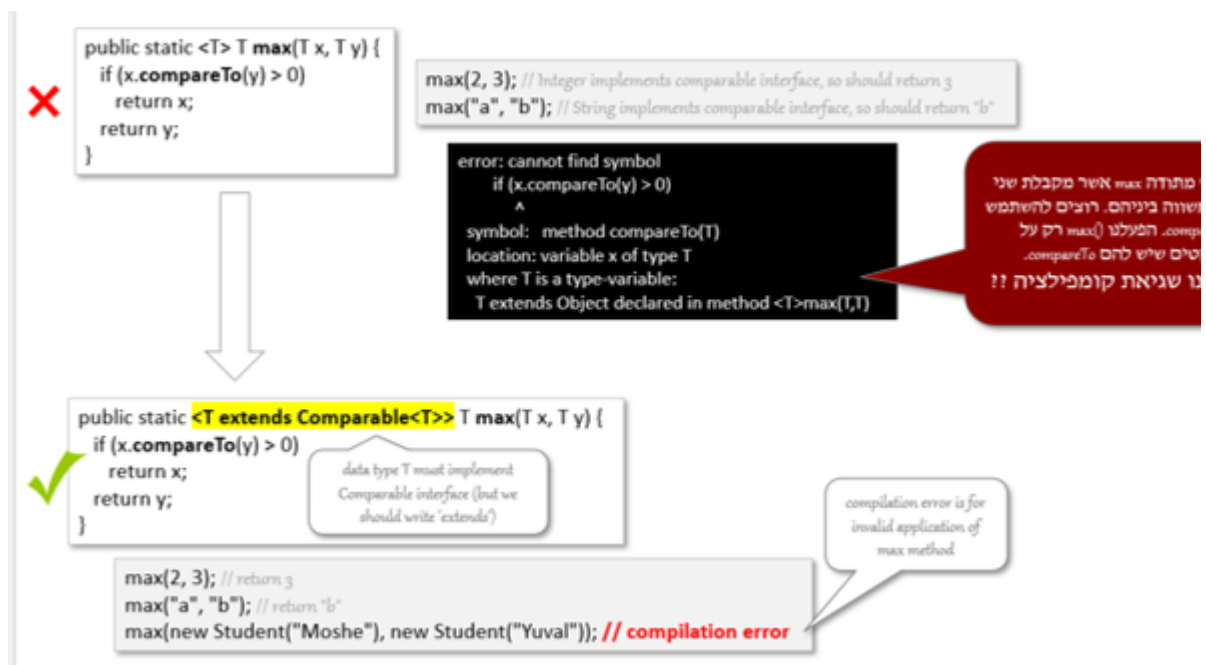
```
public static void main(String[] args) {
    Graph g = new GraphAsAdjacencyMatrix(10);
    for(int i = 0; i < g.numberOfVertices(); i++)
        g.addEdge(i, (i+1)%g.numberOfVertices());
    GraphAlgorithm algo = new EvenDegreesAlgorithm(g);
    System.out.println(algo.run()); // true
    g.removeEdge(0, 1);
    System.out.println(algo.run()); // false
}
```

## הרצאה 10: תכונות מתקדמות בOOP: טריקים שפלים בג'אווה

1. *Generics*: על פני הנייר המתודה למטה נראית תקינה, אותו שם אך טיפוס שונה. זה בסדר. כעת נסתכל על *Bad – example*. כאשר הוא גנרי נקבל שגיאה. מה השגיאה? הקומפיילר יגיד שיש להם אותו *erasure* - בזמן קומפילציה המתודות הופכות שתיהן ל-*take(Object)* ולכן יש לנו שתי מתודות עם אותו שם שמקבלות אותו דבר... וזו שגיאה.



דרך התמודדות: לשנות את השם של המתודות. לא קיימת דרך אחרת להתמודדות עם מצב זה. בכל הקלאסים הגנריים המתודות בסוף מתורגמות ל-*Object*. **דוגמה חדשה:** מתודה גנרית. בדרך לקמפל את הדוגמה מטה נקבל שגיאת קומפילציה. מהי? הקומפיילר לא מוצא את המתודה *compareTo*, מדוע? *Object* הוא לא מממש את ממשק *comparable* (בניגוד ל-*int* ושאר הפרמיטיביים), וכמו שאמרנו קודם גנריים מתורגם ל-*Object* בזמן קומפילציה. מה הפתרון? נוסיף לחתימה הרחבה של ממשק *comparable*. נבדוק בכל פעם למעשה כאשר נעשה קריאה ל-*max*. מה הבעיה? למרות שחובה לעשות מימוש ל-*comparable* ייתכן שמחלקה כמו *student* לא עשתה ואז בקריאה שלה נקבל שגיאת קומפילציה.



**תכונה נוספת על generics:** כעת נשלח אל `comparator - cmp max`, כך לכאורה נוכל להשוות ללא `compareTo`. הקריאה הראשונה חוקית. אנחנו רוצים להראות משהו נוסף מגניב - הגדרנו בתוך הקריאה של המתודה מימוש של מה?! עשינו `new interface`?! מה שקרה כאן הוא שני דברים - **יצרנו קלאס אנונימי!** אין דרך לדעת מה השם של האובייקט (מאחורי הקלעים ה-jvm נותן לו שם מוזר כלשהו), מדובר באיזשהו קלאס שה-jvm מגדיר, הוא מממש את `comparator <String>`, ולאחר מכן מממש את המתודה היחידה שיש בתוך המחלקה האנונימית הזו - כיוון שהוא מממש את `comparator`, חייבים לממש את המתודה `compareTo`. מגניב סה"כ....

```

public static <T> T max(T x, T y, Comparator<T> cmp) {
    if (cmp.compare(x, y) > 0) return x;
    return y;
}

public class StringComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}

max("Hi", "Hello", new StringComparator());

max("Hi", "Hello", new Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});

* max("Hi", "Hello", (s1,s2) -> s1.length() - s2.length());
  max("Hi", "Hello", (s1,s2) -> s2.compareTo(s1));

```

**2. lambda - קיצורי דרך.** ניתן לעשות → בג'אווה. מה מטרתו? נראה דוגמה:

```
max("Hi", "Hello", (s1,s2) -> s1.length() - s2.length());
```

זו כתיבה הכי מקוצרת שיש בג'אווה כאשר אנחנו מגדירים קלאס אנונימי. מה אמרנו כאן לקומפילר? כאשר אתה תפעיל את המתודה `max`, אתה יודע שאתה צריך לקבל שני שמות וכן `comparator` כלשהו. ניתן לשלוח `comparator` מקוצר כבר בתוך החתימה. ניתן באופן דומה לעשות

```
max("Hi", "Hello", (s1, s2) -> s2.compareTo(s1));
```

מה תפקיד החץ? החל מהחץ → מתחיל המימוש של המתודה/ממשק. למעשה, את המתודה הפשוטה

```
(int a, int b) -> { return a + b; }
```

ניתן לכתוב כ

```
(a, b) -> a + b
```

לשים לב זה אפילו בלי נקודה פסיק, מדוע? התכונה מגיעה במקור מפייתון.

**הערה:** ניתן לעשות שימוש ב *lambda* רק עבור מימוש מתודה אחת במחלקה האנונימית.

**\*מתודה** *forEach*: על *ArrayList* ניתן להפעיל על אובייקט מסוג זה *arr.forEach((i) -> system.out.println(i))* כאשר היא תעבור על כל האיברים ברשימה / מערך ותעשה מה שנדרש ממנה מימין.

**נראה כי הערות בשורות הכחולות הינם למעשה מחלקה אנונימית! זהו המימוש של המחלקה *run* שמופיעה למעלה. איך יודעים זאת? לפי החתימות! אם היה עוד חתימות בתוך *stringFunction* לא יכולנו לעשות את השימוש ב *lambda***

```
public interface StringFunction {
    String run(String str);
}

public class Main {
    public static void main(String[] args) {
        StringFunction exclaim = (s) -> s + "!";
        StringFunction ask = (s) -> s + "?";
        printFormatted("Hello", exclaim);
        printFormatted("Hello", ask);
    }
    public static void printFormatted(String str, StringFunction format) {
        String result = format.run(str);
        System.out.println(result);
    }
}
```

למעשה Java בנתה מאחורי הקלעים שתי *anonymous classes*, *class* עבור כל הגדרה של מטודה *run*. בנוסף להגדרת מחלקות, Java בנתה שני אובייקטים, אחד מכל מחלקה. החפניות *exclaim* ו-*ask* מצביעות לאובייקטים אילו.

Hello!  
Hello?

**דוגמה מבלבלת:** האם הקוד הבא תקין? הסוג של *obj* הוא תמיד *IA*, לממשק יש שתי מתודות. לאחת מהן יש מימוש דיפולטיבי (ריק), ולכן מותר להשתמש ב *lambda* גם עבור מקרים כמו אלו כאשר המחלקה מממשת בפועל רק מתודה אחת (ואכן ממומשת כאן אחת בדיוק)!!



```
public interface IA {
    public int f();
    public default void g(int x, int y) {}
}
```

```
public class Main {
    public static void main(String[] args) {
        IA obj1 = () -> 1 + 2;
        IA obj2 = () -> 2 * 3;
        IA obj3 = () -> 4 / 5;
        System.out.println(obj1.f() + obj2.f() + obj3.f());
    }
}
```

**\*\* חשוב לזכור -** כאשר עושים *lambda* בתוך לולאת *for*, המחלקה שתיווצר תהיה זהה לכל הלולאות. לעומת זאת, כאשר מחקים את רעיון ה-*for* ומשכפלים קוד 3 פעמים יוצרו 3 מחלקות שונות.

**\*\* חשוב לזכור -** כאשר כותבים *getClass* נוצר אובייקט. **3. *lambda* עבור מחלקות אבסטרקטיות:** נרצה להבנות איזשהו *lambda* מ-*abstract* מחלקה. הדוגמה משמאל לא תעבוד ותחזיר שגיאה. הדוגמה מימין כן תעבוד - עם מחלקה אנונימית.

```
abstract class A {
    public abstract void f();
}
class Main
{
    public static void main(String[] args) {
        A a = () -> System.out.println("Hi");
        a.f();
    }
}
```

error: incompatible types: A is not a functional interface  
A a = () -> System.out.println("Hi");



```
public abstract class A implements IA {
    public abstract void f();
}

public class Main {
    public static void main(String[] args) {
        A a = new A() { // Anonymous class instance
            public void f() {
                System.out.println("Hi");
            }
        };
        a.f();
    }
}
```

**4. *reflection*:** יש לנו מחלקה ואנחנו מעוניינים לקבל מידע עליה. ראינו שאנחנו יכולים לעשות זאת עם *getClass*. כעת נתבונן ונבין מהדוגמה מטה - מסתבר שיש פקודה *DeclaredFields* שמחזיר *collection* של כל השדות, לכן ניתן להפעיל עליו *length* ולדעת כמה שדות יש. גם יש מחלקה בשם *Field* מסתבר. מסתבר שבאופן דומה יש פקודה שמחזירה רשימה של המחלקות - *DeclaredMethods*. וגם כאן יש מחלקה בשם *Method*. יש פקודה בשם *getSuperClass* - היא מחזירה *null* על *object* בלבד, ועל כל מחלקה אחרת היא תחזיר את המחלקות שיוורשים מהם. **כאשר נדפיס את כל המתודות כמובן שנדפיס גם את המתודות של כל מי שאנחנו יורשים מהם - ואנחנו תמיד יורשים מ-Object!** זה הורס *encapsulation* שכן כל ההגנות נשברות ויכולים לשנות תכונות מסוימות. *native* - מתודות שלא ממומשות ב-*java* ולוקחים אותם מ-*c++* או *c*. לא נרחיב כרגע ולא כזה חשוב. (לא למבחן)



```
class A {
    private int aInt;
    private String aString;
}

class B extends A {
    private double bDouble;
}
```

```
public static void main(String[] args) {
    print(B.class);
}
```



```
public static void print(Class c) {
    System.out.println("class: " + c.getName());

    if (c.getDeclaredFields().length > 0) {
        System.out.println("fields: ");
        for (Field f : c.getDeclaredFields())
            System.out.println("    " + f.getName() + ": " + f.getType());
    }

    if (c.getDeclaredMethods().length > 0) {
        System.out.println("methods: ");
        for (Method m : c.getDeclaredMethods())
            System.out.println("    " + m.getName() + ": " + m);
    }

    if (c.getSuperclass() != null) {
        System.out.print("inherits from: ");
        print(c.getSuperclass());
    }
}
```

```
class: B
fields:
  bDouble: double
inherits from: class: A
fields:
  aInt: int
  aString: class java.lang.String
inherits from: java.lang.Object
```

## 5. Garbage – Collector

לא נעמיק בדיון ובטח שלא נדבר על איך שהוא בנוי. ניתן להפעיל אותו באופן ידני מתוך התוכנית - אם כי זה לא מומלץ! איך קוראים לו? `system.gc()`; קורא לו והוא יעשה מה שצריך לעשות.

## 6. Enum

כמו שראינו בשפת C ניתן להגדיר `enum`, החידוש הוא שכאן ניתן להוסיף פעולות עליו שכן הוא ממש מחלקה. לדוגמה -

```
public enum WeekDay{
    SUNDAY("sunday")
    MONDAY("monday")
    ....
    private String str;
    private WeekDay(String str)
    {
        this.str=str;
    }
}
```

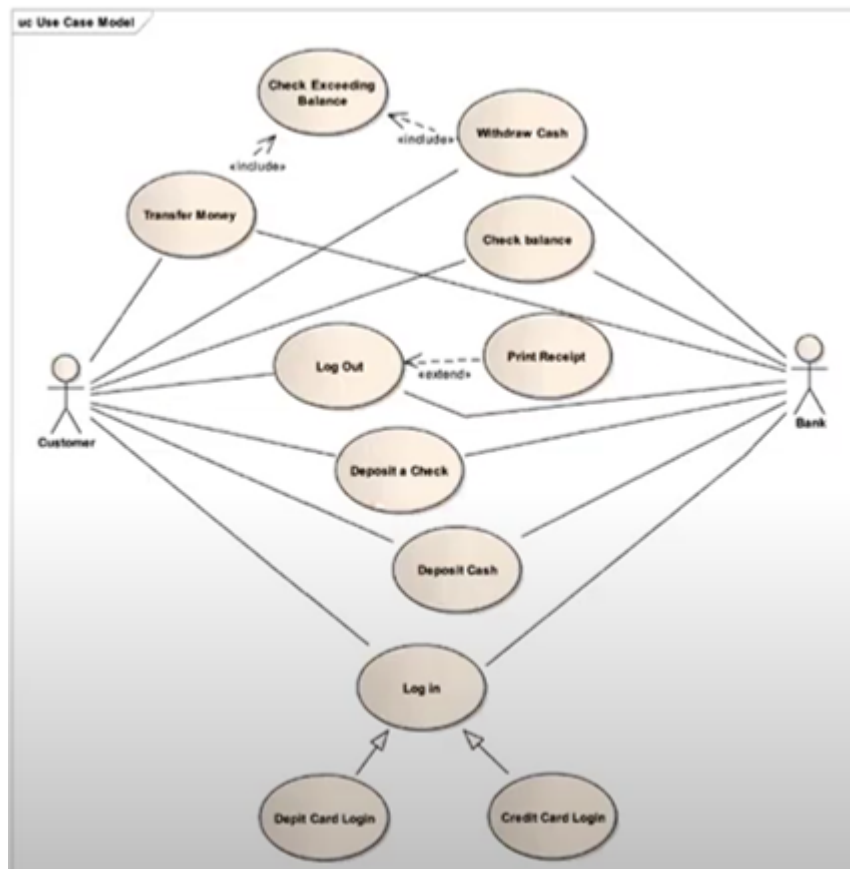
7. **UML**: הדרך שלנו לכתוב דרכים שיעזרו לנו להבין כיצד נראה הקוד שלנו (למשל הדרך עם המלבנים והגרף של הקשרים בין המחלקות). כבר בתכנות בחוץ עובדים עם **UML** שעוזרים לעשות סדר בפרויקט גדול. נצטרך להבין אילו אינטרקציות קיימות. למשל במערכת של בנק: כאשר אני מושך כסף - מי מעורב בזה? חשבון הבנק, הבנק, מערכת הפריטה. כאשר אנחנו ניגשים לפרויקט, עוד לפני שניגשים לתכנת, פועלים בשלבים הבאים:

- חושבים מה הדרישות שיהיו מהמערכת, לרוב הדרישות יגיעו מהלקוחות שלנו. אם פרויקט קיים ומשפרים - אילו שיפורים לקוח רוצה ממני לבצע.
- לפני לתכנן: מיהם השחקנים במשחק, איזה מחלקות צריך עבורם, איזה פעולות אנחנו רוצים שהמערכת תבצע?

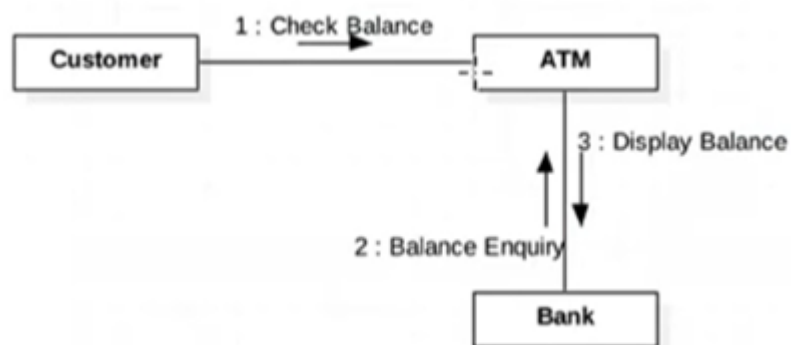
- לתכנן: לסדר את המערכת לפי מחלקות והורשה, ולעשות כמה *uml* שונים.

ה**UML** מתחלק לשניים. יש כאלו שמגדירים את המבנה (כמו זה שראינו) ויש כאלו שעוברים על ההתנהגות. מדובר בסה"כ בתיאור ויזואלי של המערכת.

1. **דיאגרמה של התנהגות: use – case**: הדיאגרמה לוקחת אובייקטים שבמערכת, ומסבירה עם חצים מה הקשר בניהם מבחינה התנהגותית. לדוגמה -



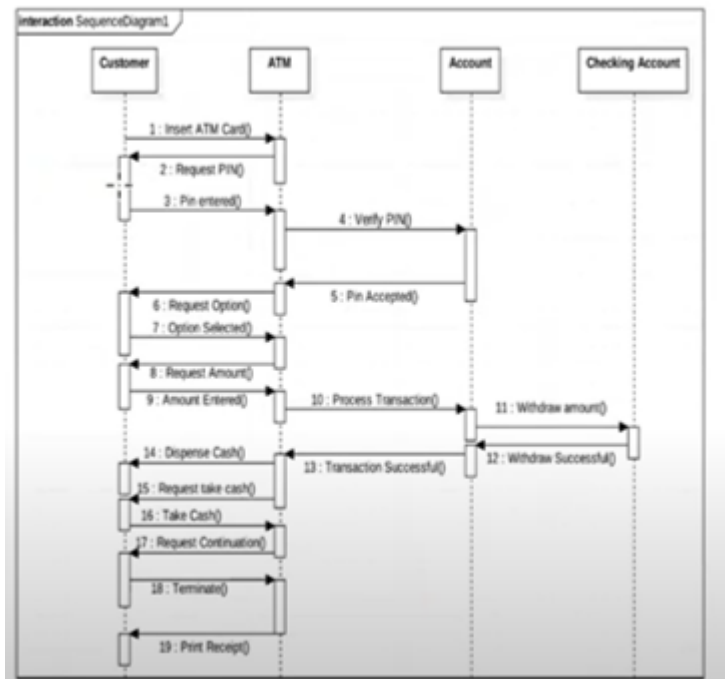
2. **דיאגרמה נוספת נקראת** *diagram communication* שתפקידה הוא להסביר את הקשר בין המחלקות השונות בהפעלת פעולה. למשל בדוגמה מטה - הלקוח מפעיל את הפעולה, משם זה הולך ל-ATM, ואח"כ לבנק. ההבדל בין זו לקודם הוא שמקודם ראינו את החיבור בין העצמים, וכעת אנחנו רואים את הסדר שקורה בזמן הפעולות.



3. *state - diagrams*: כמו במודלים בתיכון, יש אוטומט כלשהו ובו מתואר הפרוייקט. כמובן אם יש לולאה עצמית אז נשארים בתוך המצב עד לשינוי, וכן ניתן לעבור ממצב למצב בהתאם לדרישות הפרוייקט שלי. כאשר כל אחד מהמצבים לא מצליח, הוא ילך למצב כשלון ששם הוא יעשה מה שיצטרך לעשות..

4. *Activity diagram*: קודם לכן הגדרנו תהליך שלם, כאן אנחנו נגדיר פעולה ספציפית שמחולקת למחלקות השונות, ויש חצים מקשרים כמובן.

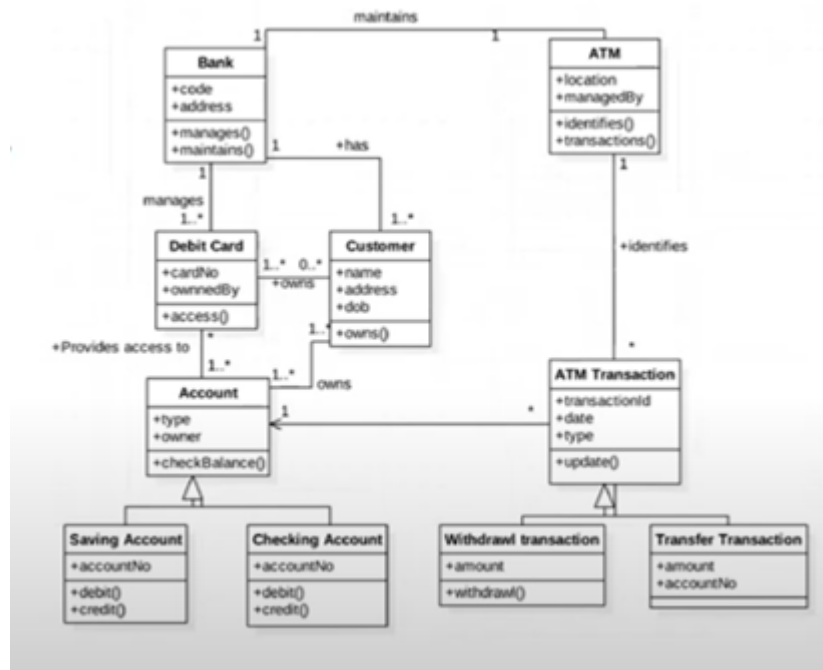
5. *Sequence diagram*: דיאגרמה מאוד חשובה לפי צביקה. מציקה גם את התקשורת בין הפעולות. ניתן לראות ממש את החצים של התהליך בכמות זמן, כמה זמן יתבצע הפעולה, איזה פעולות יתבצעו תוך כדי? המלבנים מייצגים את הזמן גם כן. אחת הדיאגרמות החשובות.



המסקנה מכל הדיגארמות היא שבכל פרויקט נשתמש בכאלו, ובכמה כאלו.

## דיאגרמות של מבנה:

1. *class diagram*: הנפוצה שראינו. תיאור של הפרויקט: איזה מחלקות, איזה פעולות יש בהם, מה התכונות של המחלקה, מה הקשר הורשה/ אינטרפייס וכו' בניהם. חץ מדבר על ירושה, חץ מקוקו מדבר על ממשק, המספרים 1 מתאר כמה יכול להיות, למשל לכל בנק יש *ATM* אחד, אבל לכל בנק יכולים להיות הרבה *costumers*. הרבה מסומן ב\*. לפעמים כתוב על היחס משהו - מה היחס, לא חובה. מה זה המעויין? מעויין ריק - מצייין הכלה, מעויין מלא מייצג "קשר חזק". למשל - שיעור לא חייב להיות תלוי בסטודנט - יתכן שיעור ללא סטודנטים. אבל יש קשרים כמו תלמיד, ותלמיד למדעי המחשב. לא יתכן תלמיד למדעי המחשב שאיננו תלמיד. חץ מצייין הורשה.



סימנים של *class - diagram*:

\* - : *private* = + , *public* , # - *protected*.

יהלום - הכלה משמעותית, יש תלות של מחלקה במחלקה אחרת (למשל המחלקה היא שדה שלה)  
קו מקוקו - משתמשים במופע של המחלקה השנייה באחת המתודות, אך אין תלות

8. *TDD*: *test - driven - devolpment*: עשית דיאגרמה. כתבת אחלה קוד. הרצת כמה פעמים.

מצאת באג. תיקנת בבאג. כתבת טסט, עבד. סבבה. כאן זה הפוך - קודם נכתוב טסט. נכשל  
בו. כעת המטרה לפתח את הקוד שלא נכשל בו. המטרה הפוכה: כעת אנחנו מנסים להתכונן  
לטסט, ולא סתם על עיוור ואז להכין טסט. בכל פעם אנחנו נבצע טסט, אם נעבור אותו נתקדם  
לטסט הבא. אנחנו בכל פעם נריץ את הטסט הנוכחי ואת כל הקודמים - וזה מאוד חשוב, שכן  
אם טיפלנו בבאג אחד יכולנו ליצור אחד אחר. לכן זה יבטיח שזה לא יקרה.