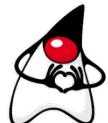


תכנות מונחה עצמים - סיכום הרצאות

22 ביוני 2025



- * הסיכום נכתב תוך כדי השיעור ויתכן שנפלו בו טעויות, על אחוריותכם בלבד.
- * שימו לב שיש בעיה עם האנגלית ולפעמים נכתב הפוך - קראו הפוך
- (c) גיא יער-און

הרצאה 1:

דברים כלליים:

- אם יש לי קובץ הרצה במחשב שלי, ואני להעביר אותו למחשב אחר. האם בהכרח ירוז? לא, ואפלו נראה שלא. זה תלוי בהרבה דברים ובעיקר באותה ארכיטקטורה ואם יש לי את אותה משפחה של מערכות הפעלה.
 - תמיד ב-*java* המופעים של מחלקות - למשל מופיע של המחלקה *point*, יופיע ב-*heap*.
 - נעשה כשרצחה לפנות לאחד השדות מתוך המחלקה, זו לא חובה אך הרבה יותר קריא *this*.
ומאפשר מצבים של *this.xPoint = xPoint*. בשדות המחלקה כموבן שלא נכתב *this*.
 - ב-*java* אין מצביעים! יש *Pointp = newPoint()* - למשל *Pointp* references *new* המילה השמורה *new* היא רפרנס. מאחרורי הקלעים יש עבודה *malloc*. *new* יחזק את הכתובת של האובייקט החדש שיצרנו.
 - בקורס שלנו, *class* לרוב יהיה *public*.
 - אסור בתכילת האיסור לגשת לשדות מחוץ ל-*class*, זה יגרור שגיאת קומpileציה. מה כן?
getters ו-*setters*.
 - השדות והMETHODS יחד יקראו *classmembers*.
- * נניח ועשינו פעולה *p = newPoint(2, 3)* וכעת נרצה לעשות *Pointp = newPoint(5, 6)* זה יהיה חוקי!
אין שום שגיאה, ה-*garbagecollector* יטפל בזה ו"ינקה" את הערכים הקודמים - זה רק יעלה לנו ביעילות.

מחלקה :String

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

- ***פעולה** (*indexOf(char)*): מוחזיר את האינדקס משמאלי של *char*, אם לא מוצא יחזיר -1.
- יש לשים לב שב *java* המחלקה היא אובייקט, ובדומה גם מערכיים הם אובייקט (נעיר בהמשך)
- שני פעולות מיוחדות ששויה להתעכ卜 עליהם הם *s.equals(s)*. הן שתי פעולות שקיימות בכל מחלקה ויש לדרוס אותן ולמש עצמן כיון שהערך הדיפולטי שלהם לא שווה לכלום.
- **הפעולה** (*s.equals(s)*) בודקת האם הרפרנסים זהים ולא התוכן - מבלבל מאד ויש לשים לב לזה. למשל עבור הדוגמה הבאה:

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

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

```
[
```

נקבל שיוודפס סה"כ *false* שכן אמנים התוכן זהה אך הרפרנסים שונים! זה לא מה שציפינו לקבל וכלן נדרוס.

הפעולה (*toString*) מבדיקה "האש קוד" מזהה לכל אובייקט והוא מדפסה ושוב לא בהכרח יודפס תיאור כפי שחשבנו - גם אותה נדרוס.

תכונת : *Encapsulation*

- בואו נסתכל על דוגמה שגوية:

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

קונסטרוקטור זה לא דבר פשוט לפעמים, כאן זה שגוי למורי. נשים לב שהוא אכן יתקملפ אך יש בעיה בעקרונות של *oop*. נוכל לעשות דרך *main* *p.setX(p.setX* וזה ישנה אכן את ערך הנקודה מה הפתרון? לשנות את השורה השנייה ל -

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

ובעת זה מועלה לנו.

למה שראינו כאן קוראים *encapsulation*. נראה כי קיימים מצבים רבים בהם נרצה להגן על השדות שלנו כי *private* לא תמיד יעוז, ממצבים כמו חובה להמנע. בהתאם לכך גדרים וסטרים גם כמו בדוגמה כאן זה יותר מלביגיטימי.

- יש לשים לב שמחלקת *random* כאשר משתמש בה בתוכנית, יש לשים אותה כחלק מהשדות.
- בתוך *class* שונים, ניתן לקרוא למתחדות בשם זהה.

הרצאה 2:

תכונת : *Static – members*

- סוג מיוחד של *.class members*

מווטיבציה: לספר כמה פעמים לאורך הפרויקט יצרנו *point*. אם ניצור משתנה רגיל שיהיה חלק מהשדות של המחלקה, בכל פעם הוא יתאפשר מחדש כשןכנס למחלקה. לעומת זאת אם נכתוב את השורה הבאה -

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

cashdug הוא על *static*, וכך נצליח לספר את מספר המופעים. זו תכונה ששhicת למחלקה ולא לאובייקט!

השדה סטטי הוא שיך לקלאס, ואתחול יבוצע 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.

cashracha לגשת למוחודה סטטית, נעשה זאת דרך השם של המחלקה.

* ניתן לגשת למוחדים סטטיטים/ מוחודה סטטיט גם דרך אובייקט של המחלקה. אך זה לא נכון! זה יתאפשר אך לא יפה (ולא יפה - לא עושים).

* מוחודה סטטית - דוגמה לא טובה: אם נעשה

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

זה כמובן לא יתאפשר. מי אלו בכלל איקס וויאי? אין גישה אל משתנים שallow הם שמותיהם.

ספוריית *math*: הכל בספרייה הוא *static*. נשאלת השאלה - מדוע יש במחלקה מוחודה שככל לא משתמשות בתוכנות שלה? התשובה היא ש*java* חובה להגדיר כל מוחודה בתוך מחלקה כלשהי, וזה המחלקה ההגונית ביותר להכניס אליה את המוחודות הנ"ל.

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

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

```
[
```

כאשר *g* היא מוחודה במחלקה. כלומר - חוקי לעשות *new* לאותה מחלקה מתוך אותה מחלקה. באשר לדוגמה - היא חוקית כיון שנדרש ליצור מוחודה אובייקט מתוך מחלקה, בעת נזמן את "she" שיהיא *a* ואת *g* אין בעיה להפעיל על אובייקט.

* מוחודה סטטית, לא ניתן לגשת אל מוחודה לא סטטית. כיון שמוחודה סטטית שיכת למחלקה בזמן שמוחודה שאינה סטטית היא ברמת האובייקט.

* אם השתמש הרבה *static* נראה שלא הבנו נכון את עקרונות *oop* ונדרש למכת חזרה אל המצביעים והכיף ב*c*. צריך להצדיק שימוש במוחודה ומוחדים סטטיטים! נראה שהרנו שפה שאינה נכונה לפרויקט. הצדקה היא מקרים כמו הקאונטר מעלה.

ממשק - "מי רוצה להתmeshk?"

ממשק: "רב צורותיות" - משה הוא סטודנט. הוא גם אדם, הוא בן זוג ומילואימני. בערת אליו מוחודות נתmeshk עם משה? משה? משה יתmeshk איתנו נראה כסטודנט. עם המפקד שלו, יתmeshk משה כמילואימני. לאובייקט כזה נראה פוליפורמייז.

כעת, נניח ויש לנו מחלקות ריבוע ומעגל. נרצה למש מוחודה שתתקבל ריבוע ומעגל. היכן נשים אותה? ==> מחלקה חדשה בשם *functions*! חוקי אבל בשביל מה? יש ברגע�ה אפשרות חדשה ומעניינת -

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

ולכל המחלקות שישתמשו במשמעות נוספת נוסיף `.Public class _name implements shape`

מה קורה לנו? המשך יעזר לנו והוא יהיה מעין "הצהרת כוונות" כל מי שיממש אותו יהיה חייב למסח את כל הפונקציות שהוא מצהיר עליו בתוכו, כולל מחלוקת משולש תהיה חייבות להחזיק מתודה שמחשבת שטח משולש ובדומהה עבור ריבוע וכל מי שיממש את `shape`.

כעת, `shape` הוא דטה-טיפ חדש. רעיון דומה `typedef` בשפת `c`.

cut נתבונן בדוגמה חשובה -

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

ונתבונן בדוגמה הבאה:

A. שגיאת קומפליציה - `IA a = new IA();` על מה הוצהר כאן `"new"`? אין שום `new` לעצם חדש שמשתמש בבנאי שלא קיים - שנייה כמובן.

ב. תיקון -

```
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 המשך - במשך אין מתודה `g` שכון קיימת אמן ב`A` אך לא במשך ולכן שגיאת קומפליציה.
ד.

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

חוקי ורץ - יש כאן קאסטינג והפיכת המשך לעצם של `A`, שם אכן יש מתודה `g`. בעצם מאחרוי הקלעים היה לנו בקשה מהקומפיילר - תנו לי לעשות *casting* והוא אישר לנו כי זה הגיוני אך באותה המידה יכול שלא לאשר! קאסטינג כאן הוא מסוכן מהסיבה הבאה - אם במקום `newA` היה לנו `newB` היינו מקבלים שגיאת זמן ריצה ! מדוע? נניח `A` הוא ריבוע ו`B` הוא עיגול. היינו מmirרים ריבוע לעיגול?! כמובן שלא.

הערות מתרגול 2 (צביקה ברגר)

* המחלקה `Integer` הינה `java.lang.Integer` כתוב כך - `IntegerB = 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.

* פרטיטיבי ולכן הדוגמה הבאה לא חוקית ולא תתקMpFL -

```
integer b = null
```

```
int a = b
```

המלצת צביקה - להשתמש במחלקה זו ולא ב-`int`.

*תוכונה מקורס 'תכנות מתקדם' שכן הוזכרה בתרגול: נתבונן בדוגמה הבאה -

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

באופן מפתיע יוחזר כאן אמת, תוכנה זו נקראת `StringPool`, ג'אווה מחליטה עבור מחרוזות זהות בתוכן שלhn לשמור אותו באותה הכתובת (שימושי כאשר יש שתי מחרוזות ענקיות זהות) - ולכן כיון שהכתובת כאן זהה יוחזר אמת.

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

* ניתן לקרוא לكونסטרקטור אחד מكونסטרקטור אחר - נתבונן בדוגמה:

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

מה יש כאן ומדוע זה חוקי ומתקפל? יש לי מחסנית עם שני קונסטרוקטורים: אחד שמקבל גודל ערך מחסנית מסימלי והשני ריק. את הריך נוכל להגיד באמצעות זה שמקבל - בעצם המחשב מבין מה עשינו וכיון שהשתמשנו במילה השמורה `this`, אז הוא הולך לكونסטרקטור שכן קיבל ערך וקורה מה שקרה.

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

הרצאה 3:

ירושה - Inheritance

* נשים לב שלפי עקרונות `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` וללאו שיורשות ממנה נוסיף ביצירה `extends`. אנחנו נירש כל מה שיש ב-`baseClass` ונוכל להוסיף כموון עוד ..

* **התוכונה override:** אנחנו בתוך מחלקה שירשה מחלקה אחרת מתודה בשם `f`. כתע, נרצה לדرس אותה ולממש אותה אחרת. פשוט ואפשרי - נעשה זאת כך:

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

מכאן המסקנה היא שניתן לרשות רק חלק מהמתודות ולא את כולם. נשים לב כי אם דרשונו את רוב המחלקה שירשנו ממנה, מדוע לרשות ממנה מלכתחילה? זה לא נכון!

תכונת upCasting: המרה של אובייקט ממחלקה בת למחלקה אב.
למשל - יש לי מחלקה "חיה" ואת מחלקה "כלב" שיורשת ממנה. זה "לשדרג" את כלב לתפקיד חייה. (כਮובן שזה מאד מאד נאמר לצורך הפשטות הרעיון)
תכונת downCasting: המרה של אובייקט ממחלקה אב למחלקה 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 השניה בדואן קאסט פשוט נגיד לקומפיבילר - "אתה רואה אותו עכשו כחיה, אבל פשוט תתייחס אליו כלב"

דוגמה 2 לא עובדת ונΚבל שגיאה בזמן ריצה - מדוע? גם המשטנה חיה הוא מסוג חיה אך הוא מצביע גם על טיפוס של חיה. איפה יש כלב? אין כלב כמו קודם. אנחנו מנסים להזכיר את הקומפיבילר עם דואן קאסט לומר לו - אתה כלב, והקומפיבילר חכם (אבל לא עד כדי כך..) הוא יקנה את זה בהתחלה בקומפילציה, אך בזמן ריצה יכשל וסה"כ נקבל שגיאת זמן ריצה.

נתבונן בעת 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);

63

1. נשים לב כי איןנו תקין. טיילור הוא employee והוא לא מתודה בשם *program*
2. תקין, טיילור אכן בקאנדרוגרמר ועשינו דואן-קאסט.
3. לא תקין, אין קשרABA-בן בין פrogrammer לemployee, עם זאת לפי היררכיה יכול להצביע על אובייקט שמשמש פrogrammer ולכן קומפיבילר יאשר קאסטיניג. קומפיבילר לוקח סיכון וקיבלו שגיאת זמן ריצה.
4. באופן דומה ל-3: קומפיבילר לוקח סיכון, גילה שאין מתודה כזו ושגיאת זמן ריצה.
5. שיטות גמורה.

הורשה משולשת/ ממתק משולש

* ג'אווה לא מאשרת הורשה משולשת -

* ג'אווה כן מאשרת זאת במשקיים -

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 משנה הממשקים נקבל שגיאה! זה לא חוקי להוריש משנה ממשקים עם אותם החתימות (ומשניהם קלאס בכלל אי אפשר להרחיב..)

Abstract class

יש מחלקות שלא נרצה שנוכל ליצור טיפוסים חדשים מהם. למשל *employee* היא כזו כי אין מישחו שהוא רק עובד, נרצה ליצור פרודקט מאנג'ר, פרגמר וכו'... בשביל למונע מה משתמש ליצור אובייקטים מהמחלקה נוסף לשם *abstract*. אז - אם משתמש ירצה להשתמש יקבל שגיאת קומPILE. כמובן, ניתן למחלקה מקום בפרויקט אך לא ליצור ממנה. במחלקה אstractית - ניתן ליצור מתודה אstractית למשל ()*w*. עבור כל מתודה - או שמנמש אותה, או שנשים לה *abstract*.

קלאס שמרחיב קלאס אstractiy לא חייב להיות אstractiy כMOVEN. אך מה כן? הוא חייב לממש פועלות אstractiy של המחלקה - למשל ()*w*. *אם נעשה קלאס וכל המתודות בו אstractiy זו התנחות לא נכונה (ואפס בבדיקה) => היה צריך לעשות *Interface*. לסיום מתודה אstractiy - מתודה בלי גוף - כמובן, רק החתימה שלה מוגדרת, אבל הקוד עצמו לא קיים. נמש את הפעולה עצמה במחלקה יורשת.

הפעולה *super*

אנחנו נמצאים במחלקה משולש שווה שוקיים. היא יורשת ממחלקה משולש. בקונסטרקטור - נרצה לבדוק האם אכן המשולש הוא משולש. עייתי לבדוק בקונסטרקטור כי כבר יצרנו אובייקט לא? אז נשלח לקונסטרקטור נקודה עליונה ואורץ צלע והוא צריך ליצור משולש שווה שוקיים זהה. 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 מחלוקת האב מתוך מחלוקת הבן, עם המילה סופר נקרא גם לבני של מחלוקת האב בשורה הראשונה בלבד! תכונת נספת מעניינת - ניתן להשתמש במתודה הקודמת לפניה שדרסנו אותה באמצעות המילה סופר: אנחנו בחלוקת הבן דורסים מתודה שקיים בחלוקת האב, נוכל להחליט שאחננו קוראים לפעולה הקודמת מתוך הפעולה החדשה שדרסנו. למשל -

```

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

```

המילה השמורה : *Protected*

תכונה שנוציאפ במקומות *private*. במחלקות שיורשות - ניתן להשתמש באופן ישיר באמצעות בתכונות של מחלוקת האב. אך! אם ננסה להשתמש בו במחלקות שאינן יורשות ממנו הרי שזו שנייה.

חשיבות מתרגול 3 (צביקה ברגע)

אוסףים - *collections*:

- יש שני סוגי אוספים. *list* ו- *set* כאשרリスト אין חשיבות לסדר האיברים (קבוצה), נטמקד בעת בעיקר בLİסט. יש שני סוגי -

1. *arrayList*

2. *LinkedList*

יש בהם הרבה מתודות בנויות שימושיות....

הרצאה 4:

מחלקה `class object`

מחלקה מובנית בג'אווה. שתי פעולות שראינו בעבר - *equals* ו- *toString* שייכות לה. באופן אוטומטי ע"י ג'אווה, קלאס אובייקט הופכת למחלקת האב של כל מי שלא יורש. עבור מחלקות שכן יורשות זה `sub class`.

ניתן ליצור אובייקט של מחלקת אובייקט - `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 שהציגנו בעבר - יש הרבה בעיות עם *equals* ו- *toString* ולכן נדרוס אותם לרוב ונשכתבם מחדש.

נתבונן בקוד הבא שילוחו אותנו:

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

במצב זה אכן הפעולה נראהיה נכונה - ככלומר ניתן לקרוא `o.equals(x)` כי זה מתוך המחלקה.
אז מה כן הבעייה בקוד?

איך נדע שאכן `object` הוא `point` ומהזיך ערכיו איקס ווואי?
`<<<<=====`

הפעולה `InstanceOf`

דוגמה תמחיש היכי 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 הבעייה! 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?

וכעת טיפולנו בבעיה. כלומר מה למדנו? ניתן לגשת עם `getClass` גם לשדות עצמן שהאובייקט מתאר, כמו שם. וכעת כמובן שם המחלקה שונה ==> לא אותו טיפוס וסימנו.

נושא 3 - Static&Dinamic biding

דוגממה הכי טובה -

```
Polygon n = new name();
```

הפוליגון - הוא הטיפוס הסטטי! `name` הוא הטיפוס הדינמי.
בזמן ריצה קומפילר בונה TABLETABLE `vtable` לכל `class` בזיכרון, טבלה שמחזיקה חלק(!!!) של הכתובות של המתודות של המחלקה.

*אם מישחו ירש - הכתובת למתודה תהיה היכן שהכתובת למתודה של האבא.
קומפילר דוחף שדה לכל אובייקט של `vptr` שייתאר את הכתובת בזיכרון.
בואו נתבונן בדוגמה של דינמיκ בינדינג -

```
Polygon t = new SpottedTriangle(..)  
t.getName();
```

מה שיקרה במצב זה הוא דינמיκ בינדינג. כאן הוא יבצע את `getName` של

לסטיק -

סטטיק בינדינג מתרחש בזמן קומפליציה - ככלمر, בשלב הקומפליציה, הקומפילר יודע בדיעוק איזו מתודה יש להפעיל. יותר משתלים בזמן הריצה.

динמיκ בינדינג - מתרחש בזמן ריצה - כאן הקומפילר 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 סוגים מותודות לא נכונות `vptr`. - פריבט, סטטיק ופינל.
זמן טוב לדבר על **מתודות פינל** - final version, לא ניתן לדרוס אותה ואם ננסה לקבל שגיאת קומפליציה. באופן דומה, class final היא מחלוקת שלא ניתן לרשף ממנה.

* עיר כי יש דוגמה מצוינת במצב 4 על הנושא בעמוד 31.
* עבור מתודה סטטית, בקומפליציה זה ילקט טיפוס הסטטי ולא הדימי.
* עבור משתני המחלוקת - 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 שללחנו.
מה אסור? מותודות 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
* דוגמיה חשובה -
יש שתי מחלקות A ו B. B מרחיבה את A.

`func(A x)`
`func(B y)`
 מה קורה במצב זה? קומפילר בוחר לפי סטטיק בידיניג, בהתאם לטיפוס הסטטי של ארגומנטים
 כפי שהסבירנו.
 לעומת זאת,
`func(A x, B y)`
`func (B y, A x)`
 כאן קיבל שגיאת קומפליציה.

הרצאה 5:

העמסת בנאים (constructors of overloading)

- * אם הגדרנו מחלקה ללא בנאים, יהיה בניאי דיפולטי.
- *
כשנבנה אח"כ מופע של המחלקה שיש בה בניאי דיפולטי מבוון שלא נשלח פרמטרים.
מדובר בכלל נרצה זאת? לא תמיד נרצה לחיב את המשתמש לשЛОוח 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 לשЛОוח ערך אבל בבדיקה בשביל זה יש לנו העמסת בנאים - שזה יראה אלגנטיבי יותר.
- *
כשנעשה העמסת בנאים - לא נרצה שהיא שכפול קוד בשני בנאים שונים, ולכן ניתן לקרוא לבניאי אחד מבנאי אחר. איך?
נעשה *this* זה 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 קרנו לבניאי אחר של המחלקה.
בעצם במבנה השני החלטנו שנרצה לשЛОוח רק שם ללא תאריך לידיה.
*モותר לעשות כמה בנאים שנרצה.

מה קורה מאחרוי הקלעים בהפעלת קונסטרקטור?

1. ראשית מזמנים זכרו עבר כל השדות של האובייקט (כולל *Vtable* ולמחלקת *object*).
 2. אחר כך נקזה ערך דיפולטי לכל אחד מהשדות של המופע.
 3. אם בתוך קונסטרקטור היה קרייה לkonstruktor אחר - נבצע קודם אותו.
 4. גם אם כתוב וגם אם לא כתוב נדרש לכתת לטפל ב*super*. (אם יש *super* נלך לעשות אותו) ואם לא כתוב שום דבר עדין אנחנו מחוויבים לבניית לבניית האב *Object* - *object* כלומר תמיד תבוצע קרייה ל*object* בהפעלת קונסטרקטור (גם אם לא ראיינו זאת באופן מפורש בכך) - גם אם קראנו לבניי אחר הוא יהיה אחראי לקרוא ל*object*.
 5. להמשיך לבצע את "גוף" הקונסטרקטור הנוכחי.
- *reference זה 8 בתים, *string* הוא רפרנס. כמו כן נזכר כי *int* הוא 4 בתים. כאשר מזמנים אובייקט בזיכרון למשל של *student* עם *age* ושני *strings* ייה לנו $4+2=6$ בתים ועוד 8 בתים למצבייע של *Vtable* שתמיד יש מאחוריו הקלעים ועוד בתים שיש בשדות של מחלקת *Object* (נדע בהמשך..) - < בקיצור מתחבא הרבה מאחוריו הקלעים.

דוגמה טובה: נניח ויש לי מחלוקת ריקה

{ }
וגרhc class public
ונרצה להפעיל עליה המרה לטיפוס *B* בלבד - ((*B*).*c*).*g*)()) נקלט שנייה קומפילציה. כיוון שאין שום קשר בין *B* ל-*C* הקומפילר לא יתנו ונקלט שנייה 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*)()) קומפילר יאשר כיון שעלה נייר יתכן שהם אכן מאותו טיפוס אך נקלט שנייה קומפילציה. ריצה כי בסוף *a* מטיפוס *A*.

SANITY TEST



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

בاهינתן הגדרה של *A*, סמננו את כל ההגדרות התקינות של *B*.

- 1

```
public class B extends A {  
    private int y;  
    public B(int y) { this.y = y; }  
}
```
- 2

```
public class B extends A {  
    private int y;  
    public B(int y, int x) {  
        super(x);  
        this.y = y;  
    }  
}
```
- 3

```
public class B extends A {  
    private int y;  
    public B(int y) {  
        super(5);  
        this.y = y;  
    }  
}
```
- 4

```
public class B extends A {  
    private int y;  
    public B(int y) {  
        this(5);  
        this.y = y;  
    }  
}
```
- 5

```
public class B extends A {  
    private int y;  
    public B(int y) {  
        super();  
        this.y = y;  
    }  
}
```
- 6

```
public class B extends A {  
    private int y;  
    public B(int y) {  
        super(5);  
        this.y = y;  
    }  
}
```



דוגמה 1: שגואה כי חייב להיות קריאה ל`super` ואין צו - כלומר הקומפיאיר מוסיף צו ללא ארגונומטיים וזה שגיאות קומפליציה כי לא קיים ל`A` בנאי ללא ארגומנטים.

דוגמה 2: חוקי כמובן.
דוגמה 3: נשים לב שחוקי כיון שモתר בהורשה שהיו שני קונסטרקטורים שמקבלים אותו מס' ארגומנטים.

דוגמה 4: לא חוקי כיון `this` קורא באופן רקורסיבי לכואורה לעצמו - ולפי `java` לבנאי אסור לקרוא לעצמו, וכן גם אין קריאה ב-`super`.

דוגמה 5: לא נכון כיון שכמו ב-`B` קראנו ל`super` ריק ונקלט שגיאת קומפליציה כי אין 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 שתמיין אותם. לפי מה? שאלה מעולה - לפי מרחק מראשית הצירים, לפי מרחק מציר `x`,..., יש הרבה אפשרויות. באופן עקרוני על הנייר ניתן כתוב מתודות שונות לכל סוג מיוון - אך יש כאן שכפול קוד זה לא נכון ב-`java`!

*הערה: בנוסף `toString`, `equals`, `compareTo` מעתה למש גם את המתודה `compareTo` מחדש.
בדומה ל-`C`, יוכל ליצור פונקציות גנריות.
כיצד נממש?

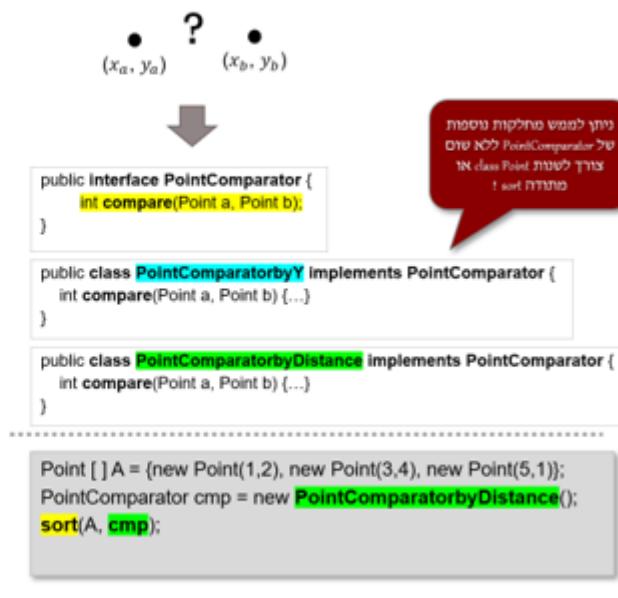
GENERICSS

```
public class Point {
    private int x, y;

    Point(int x, int y) {...}
    public int getX() { return x; }
    public int getY() { return y; }

    public int compareTo(Point other) {
        if(x > other.x) return 1; //other is smaller
        if(x < other.x) return 0; //points are equal
        return -1; //other is bigger
    }
}
```

```
public static void sort(Point [] A, PointComparator cmp) {
    int n = A.length;
    for(int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (cmp(A[j], A[j+1]) == 1) {
                int temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
}
```



* ניצור ממש `interface` שיחזיק במתודה הכללית `compare`. ואז אנחנו נחליט לשולח בכל פעם השוואה אחרת ל-`PointComperetor` שעלה פיה לעבוד ההשוואה. נשים לב שכן כל סוג מיוון הוא מחלוקת שונה!

אך כדי שיש לנו השוואה בין שני `points` נרצה לעשות השוואה בין שני קלפים למשל - אבל מה, שוב נעשה את המשק 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 שכביר נמצא בספרייה הבסיסית של java.
 point interface Comperator <T<{}
 int compare (T a, T b);

מדובר בכלים של java שנקרו generic.

כשנממש את ה *interface* אנחנו נכתב במקום T את הטיפוס שנרצה, למשל -
 public class CCByStrength implements Comperator <Card>{}
 int compare (Card a, Card b) return a.getRank()-b.getRank();

{
 ואיך נראה הקריאה?

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

:Generic Collections

אוספים גנריים מאפשרים שימוש בתבנית ייחודית לאוסף כלשהו.
 שנרצה ליצור אוסף בלי לציין את שם הטיפוס. למשל ביצירת מערך אנחנו מחייבים לציין את הטיפוס.

ולכן הגע הנושא לעולם - יש הרבה מאוד אוספים גנריים בג'אווה... *List* < E > וכן *Set* < E > ו *Queue* < E > ועוד ...
 כשנרצה ליצור אוסף מסוים אנחנו מתחייבים כבר לטיפוס מסוימים! מתחייבים לפני קומפיאילר
 ואםakh"c ננסה להפר את זה קיבל שגיאת קומפיאציה -

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

* שאלה - מדוע אנחנו משתמשים ולא מגדירים מחלוקת כדלקמן:
public class ArrayList

private - Object[] elements = newObject[10]; private int size = 0;
 מה הבעיה כאן? הרבה - 1. אנחנו מונעים לדוחף לאוסף הזה מה שנרצה מבחינת דטה טיפ, אבל
 אפשר לדוחף הרבה טיפוסים שונים (זה לא פסול) אך זה קשה כי צריך לזכור כל פעם סוגי דטה
 טיפ, " יש חשש לקבל שגיאת זמן ריצה. לעומת - ניתן להשתמש זהה אבל יש להזהר גם בהוצאה
 וגם בהכנסת איברים.

* נתן להגדיר כך - *private Class < ? > allowedClass;* מה זה אומר? ניתן להציבו לכל מחלוקת
 מהסימן שאלה, זה סוג של דטה-טיפ *Class < ? >*

:File – IO

נתן לעבוד גם עם קבצים בדומה ל *input*.
streams זרם של נתונים מקום למקום (לא שונה מזרים של מים...). פתחו קובץ וכעת ניתן
 לקרוא ממנותו (באופן דומה לטיפות המים.... (יואב בקש 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 -

File from file = new file ("a.txt")
file FileInputStream fis = new FileInputStream (fromfile)
fis.read(); ... fis.close();
 אנחנו עוסקים על האובייקט שמקורו לקובץ ולא על הקובץ עצמו.

כשנעסה (`file.read()`) אנחנו מושכים יחידה שמורכבת `byte` של הקובץ, משכנו `byte` של הקובץ, ולאתו!
 * טוב לדעת כי בג'אווה ניתן להרחיב את טבלת האסקי ולציין גם אימוגים וכו.... בג'אווה `char` זה 2 בתים.

- * כאשר נגמרה הקריאה `file.read()` יחזיר -1.
- * בשביל לכתוב את הנאמר בהדפסה יש פקודה מיוחדת - `System.out.write(readByte());` מה פקודה זו עשויה? `stream` הוא `SystemOut` לכל דבר וNSTASHIN בבה בשביל לכתוב את הביט שנקתב - היא מדפסה רק את הביט האחרון לא כל הארבעה בתוך איןט - היא טיפה ופרטיטיבית.
- * נוסיף לקוד במין בוכתרת - `.throws IOException` - נבין בהמשך מה זה אומר.
- * ניתן לכתוב לתוך קובץ - באמצעות `FileOutputStream`.
- * תמיד נסגור את `stream` שפתחנו - אחרת יהיה `MemoryLeak`.
- * **BufferedReader**: הבהיר עובד עד שהוא קולט בקסלאש אן. הוא מעתיק ביט ביט לתוכו עד שפוגש בקסלאן אן ואז ניתן להדפיס את השורה שהוא קרא והוא מרוקן כל מה שהיא בתוכו.

תרגול 5

- * עקרון *encapsulation*: עקרונו הוא להחבריא חלק מהמידע של האובייקט מהמשתמש. ככלומר הלקוח לא צריך לדעת בין היתר כיצד מימושו את הדברים ובאיזה שדות השתמשו. מאפשר למנוע מהמשתמש לשנות את המידע כיצד שהוא רוצה, אני אחילט מה הוא יכול לעשות. ככלומר אני אומר למשתמש - תעשה ככה, אני אציג זהה יעבד (למשל פקודה `printf`) מודולריות - אנחנו יכולים לעשות קוד עם תחזקה ברמה גבוהה. מאפשר החבאה של מידע. *כמובן שאסור לגשת למשתנים פרטיים ואם ננסה לקבל שגיאה.
- עקרון *Refactoring*: בינוי מחדש קוד קיים שהפוך קל יותר לתחזקה, גמיש ותקין. במקום לעשות `else if` מלא פעמים כדי חזרים על עצם - NSTASHIN בעקרון זה. נגיד מחלוקת בהתאם לכל מקרה פרטי...
 * **הערה חשובה** (זה ממש הופיע **בשאלה ב厰**): לא יתכן שיש לנו מ-variable שהיא static abstract מ-variable? מ-variable אבסטרקטית היא צו שניתן לדפוס אותה ולמש מבוחץ, מ-variable סטטית היא של המחלוקת ולא ניתן לדפוס אותה - לכן סה"כ לא יתכן אחת שהיא גם ווגם!

הרצאה 6 - *Exceptions*

- חריגת ודים יוצאי דופן. הגדרנו מושג בג'אווה שהוא תיקן אך לא נכון - אנחנוCut נדבר על מצב של **לא תיקין**.
- * יש אפשרות בג'אווה לבדוק האם קובץ קיים. אוקי בדקתי והוא קיים - מי אמר שרגע אחרי הוא קיים? אולי מישחו מחק אותו צו? נחלק את הנושא לשניים - בעיות בשליטתנו ובעיות שלא בשליטתנו - בכל מקרה זה יחשב לא תיקן. עליינו להתמודד עם כך בשני המקרים!
 - * מערכת הפעלה תהיה המעורבת והיא תהיה הראשונה שתடע על הבעיה. בין אם זו בעיה של חילוק באפס ובין קובץ שלא קיים. מערכת הפעלה שולחת `JVM signal` - זה איות `jvm` - תעשה משהו עם הסיגナル אחרית התוכנית תקרו. `JVM` מייצגת אובייקט מיוחד - האובייקט מסוג `error` והוא "נזרק" (ccccccc) - התוכנית שלנו יכולה לנפות את האובייקט הזה - לחשאל אותו ובהתאם לכך לטפל בבעיה שנוצרה.
 - דוגמה** - ג'אווה אומרת לנו שימוש `try` - נסו לעשות את מה שאתם רוצים, ואז תגדירו בлок של `catch` - אם יגיע כדור של `"error"` - אנחנו ננתפס אותנו. מדובר בשגיאות של קובץ לא נמצא, קובץ נמחק וכו.... יש לשים לב כי `try` ו- `catch` אין מתודות - הם 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, علينا לעתוף את הקוד שועלול לזרוק חריג catch block, ב- catch עליה סופי עבור 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 המתוודה שנרצה שתשלח שגיאה throws exception מסקנה - התנהגות האו תעזר לנו למנוע מהתוכנית לקרוס. היא תשלח הודעה שגיאה במקום להקרים את התוכנית.

* ניתן לשים כמה catch שנרצה - זה בדיק switchCase. כמו כן נזכר כי נלך ל catch הראשון שיפורע - לכן נרצה לשים את ההכי ספציפי למקרה ובסוף ובסוף את ההכי פחות ספציפי (הכללי יותר).

* יש מחלוקת שיוורת IOException ו FileNotFoundException והיא שונה לטפל במצב זה שונה כאשר הקובץ לא נמצא. אם כי יש תמייה לכך גם במחלקה המקורית). ל catch שיש מתוודה שהיא Shnitin להדפסה ולהבין מה הבעיה - וכך לחלק למקרים, כמו בדוגמה כאן - GetMessage

```

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 שעליינו תמיד לסגור את stream. כיצד נסגור אותו בתוך catch ? הרי אם נסגור בסוף הראשון לא יוכל להציג קדימה. בג'אווה נפעיל טרי אנדר קז' בכל פעם על reader.close(). אבל יהיה כפילות קוד זהה לא לפי עקרונות oop. לשם כך בסוף כל finally-block של הקוד שמתבצע בין אם קרתה השגיאה ובין אם הכל היה תקין. הוא רץ מיד אחרי הביצוע ב catch שתפס את השגיאה, נשים שם קוד שיתבצע בכל מקרה למשל סגירת streams. במקרה אחד בדיק שfinally לא מתבצע - אם יש פקודה exit באחד ה catch לפני

```

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) {...}
    }
}

```

הבעיה בין אם קורטנה
ו- לא היה רץ מטי **exit**
ובכל מקרה, לטעמך,
בסוף תמצא ב-
קודת **finally** **exit** ב-
שאפקטן אליהם.

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

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

משתמשים בתוכנה 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 הזכור של כל המתוודות. ניתן להדפיס למון דיבוג *stackTrace* שידפיס את כל המתוודות מלמטה למעלה בסדר הפעלתן וنبيין היכן יש בעיות. ככלומר *main* יודפס הכי למטה.

מתוודות שזרוקות *exception*: מי שקרה למתודה צו הוא חייב לטפל בשגיאה וליצור לה *tryCatch* ושם בקריאה למתודה יטפל בה ואם לא יטפל בה תהיה שגיאת קומPILEZA. במתודה עצמה לא יהיה טיפול וזריקה של אקספשן. האם זה בכלל עדיף על השיטה שראינו קודם? כן - לטפל בשגיאה עדיף לעשות היכן שקראננו למתודה, שכן הטיפול הוא פרטני. ולכן כל המתוודות של *tryCatch* בג'אווה לא יזרקו שגיאה - האחריות היא על מי שקרה למתודה זו **הדרך הנכונה** **ונפעל בג'אווה**.

ישנה בעיה - במקרה וזרק שגיאה לא נגע לשורת *close* כי בזמן *try* קיבל שגיאה ולכן לא נחזיר למתודה בכלל ולא נוכל לסגור את *!stream*? מודיע לא נוכל לסגור את הקובץ ב-*main*? הרפרנס לקובץ הוא לוקלי של המתודה שקרה לו - ולכן בחוץ ב-*main* הקובץ לא מוכר. המימוש לפתרון הבעיה יהיה כדלקמן -

```

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 כלהי בדרך main של catch לאחר catch תמיד הולכים finally וナルך לשבתו המתוודה, ונוכל לסגור את הקובץ. באופן שկול את אותה הבעיה ניתן לפתור באופן קצר יותר ויזואלית (אך מאחריו הקלעים קורה אותו דבר) כך -

try(BufferedReader reader= new BufferedReader(new FileReader("a.txt")))
 לשיטה זו קוראים tryWithResource - ימנע מאייתנו לכתוב finally לחינם וזה עשה את זה במקום מאחוריו הקלעים.
 השתמש מעכשו רק בהז סגור קובץ - כאשר נשים בתוך סוגרים של try הם יסגרו אוטומטית לאחר מכן - מבלתי שנצטרך לעשות finally כמו בדוגמה לעיל.
 * **יש לשים לב** - נניח ויש לי f ששולחת שגיאה ויש g שקראנו בתוכה lf עבור g יש שתי אפשרויות: או להגדיר בתוכה tryCatch או להגדירה כשולחת שגיאה.

סוגי שגיאות:

יש סוג בעיות שניתן לתפוס כמו כאן -

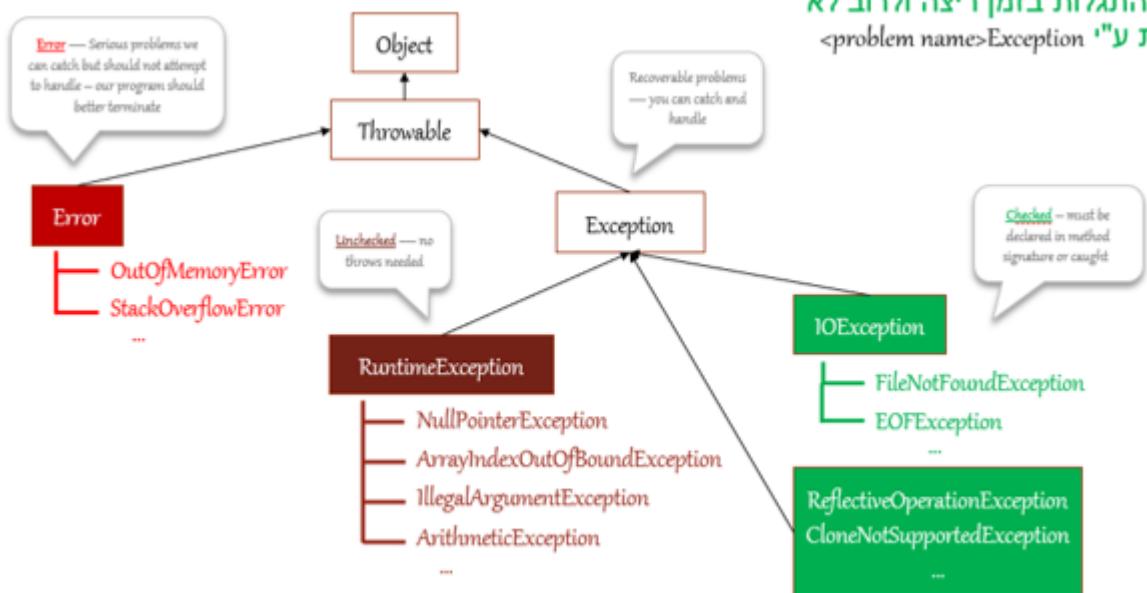
```
int x, y; ...
try { // code that may throw a ArithmeticException
    n x = x / y; }
catch (ArithmeticException e)
{ // handle the exception here System.out.println("I caught it!"); }
```

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

היררכיה השגיאות בג'אווה

נזהתנות בזמן ריצה וזרב לא גות ע"י

<problem name>Exception



- דבר שני ניתן לתפיסה בדרך "כ". לעומת *Exception* מתחלק ל-2 . 1 זה העולם הירוק - הקומפיילר מחייב אותנו לתפוס אותם בעצמי אחרית הוא יкус מאד ולא יתן למפל. לעומת זאת זאת העולם האדום - אלו שגיאות שהן שגיאות זמן ריצה - כל הבאגים .. חלוקה באפס, גישה לאינדקס מחוץ לגבולות מערך וכו'... אותנו קומפיילר לא מחייב לתפוס בעצמי. דוגמה לerror של catch - אפשר כדי להדפיס את השגיאה ואת stacktrace, ומומלץ מיד לסייע את ריצת התוכנית.

דוגמא: מה יקרה כאן? קיבלנו $i = 2$ ונתרכז לשЛОח חוזה את המתודה שלמעלה - שהוא כמובן *UnexpectedValue*. הערה - כאן כיון שיש הרחבה של המתודה *runTime* שם אין חובה לתפוס את הבעיה - לא היינו חייבים לעשות כן.

```

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());
    }
}

```

דוגמיה נוספת:

מה קורה כאן? בג'אווה החלטתו ש `main` יכול לזרוק `exception` מי שנמצא מעל `main` הוא `JVM` - הוא תפס את השגיאה ולפנּו התוכנית תקרוּס!

הערות חשובות מתרגול 6:

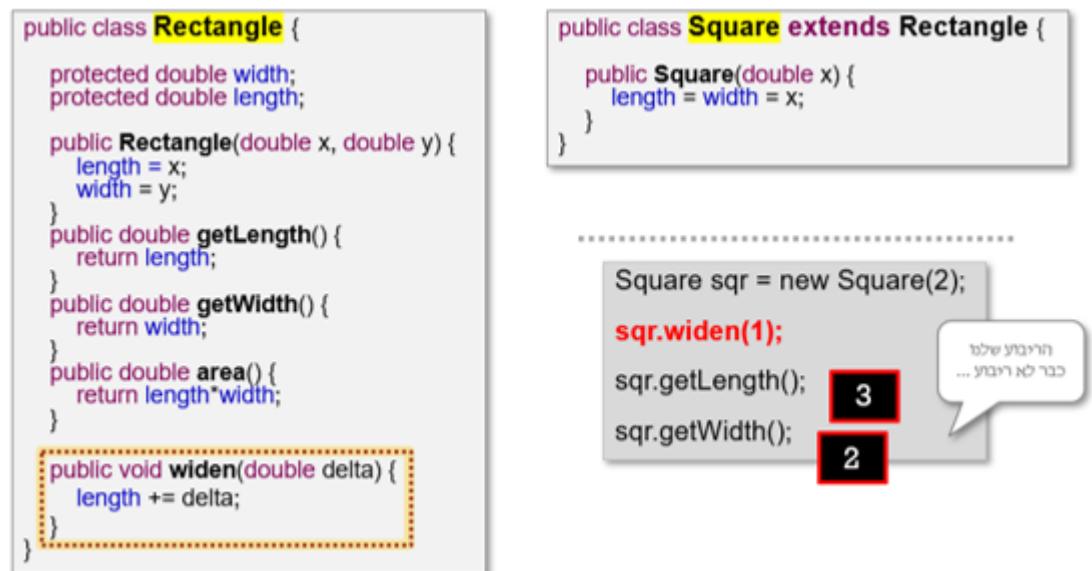
- * כל סוג החריגות יורשים מהמחלקה `exception`.
- * `check` על המשתמש לטפל בשגיאות. `uncheck` לא מחויבים בקוד לטפל בהם. `error` קיים, לא נתעסק - למשל: מחשב קרס, אין מספיק זכרון במחשב.
- * `StackTrace` - הودעה על המסך שתוצאה כאשר יש שגיאה, היא שולחת "קישורים" והפניות לשורה בקוד הנוצרה החריגנה.
- * הקשטים יכולים להרחיב מאוד - במקרה אנחנו קוראים בippet אח"כ הבנו שנוכל לקרוא `char` ובהמשך הבנו שנוכל לקרוא `line`. חשוב לציין שבפועל שימושם את הבادر רידר מאוחר הקליעים נקרא בippet.
- * יש הרבה הרחבות לטבלת אסקי - למשל `utf8` שמאפשר שימוש באימוג'יז עברית סינית וכו'.... זה כבר לא ברמת הביט.

הרצאות 7 - *:Design – patterns*

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

הערה: מופיע בסוף הרצאה 8 כאן בסיכום סי כום של **תבניות העיצוב** - מומלץ. יותר יעיל מלקראות מה כתוב כאן שכן יש בסה"כ להבין את הרעיון.

נדמיין מצב בו נkeh סטודנט שכביר 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. (לסייטואציות מסוימות). דוגמה יותר רלוונטיות: יש פרויקט עם ריבוע ומלבן - כיצד נממש? כל ריבוע הוא בפרט מלבן - לכן ניזכר מחלוקת שתקרה מלבן וניצור מחלוקת שתרש מלבן והיא תהיה ריבוע - מדוע? כל ריבוע הוא מלבן. פשוט נגדיר שבמחלקה ריבוע מתקיים *width = height*. על פניו, זה נראה טוב - זה לא! מדוע? לפי עקרון ההורשה - אם *B* מרחיבה את *A* - צריך שכל הקשר בו משתמשים בטיפוס *B* יוכל להשתמש גם ב-*A*, המתוודה הביעית היא *widen* - כשבפועל *widen* על ריבוע נקבל שהריבוע יופיע 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 שימושך עבשו מתאים לי** ואם כל מה שימושך **לי בעתיד יותאם לי**. כאן כMOVIN זה ממש לא המצב.



(הערה - יש שנייה במצב של מרינה צריך להיות *super* בתוך הקונסטרוקטור של ריבוע) אז מה הפתרון? ממש לא נשכפל קוד - לשם כך בא *delegation*. ריבוע כאן לא תרחיב את מלבן - אלא מלבן יופיע להיות שדה של המתוודה. 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;
    }
}

```

במוקם ! ext

```

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();
    }
}

```

Square sqr = new Square(2);



הוספה של המתודה
Square widening for
class Counter

כעת - אם נרצה להפעיל על ריבוע Method widen - נקבל שגיאה - כיון שמתודה זו באפנ' ישיר ממש לא נמצאת שם. ככלומר - אפשר להפעיל את המתודות שנמצאות בלבד בתוך המחלקה שלי. ככלומר כאן אנחנו נבחר מה לקבל - בנגדוד להורשה.

דוגמה עם עצ' ביןארי ל-delegated

*נרצה לדעת כמה הכנסות בוצעו לתוכו עצ' ביןארי, כאשר אנחנו מכניםים ערך לעצ' *counter* גדול וכאשר נמחק מהעץ הוא לא ישתנה - נספור כמה הכנסות בוצעו סה"כ. על פניו - נוכל להרחיב את המחלקה שלנו למחלקה "עצ' עם מונה" כאשר נוסיף שדה של *counter*. הורשה נראית מתאימה מאוד! נוסיף כموון שינוי ל-*add* עם *override add* אשר אנחנו נוסיף לעצ' אנחנו נעדכן *++counter*. מה הבעיה?

ציפינו לקבל הדפסה של 4. מה בדיק קרה לנו ??

בדוא פוקוב אחותי תקוו ורבבו

```

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

```

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);
}

```

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

מדובר קיבלנו 7 ? שלב לפני האחרון אכן $counter = 4$, אך יש קריאה של `super` בסוף `addAll`? בתוכן `addAll` אנחנו נכנסים אל `add`. `add` של מי? של העז עם מונה! בכלל דינמי בידיניג. במקרה מסווג זה של `void` הקריאה מותבצעת לפי דינמי בידיניג. איך נפתרו את הבעיה? פתרון אחד - אפשר למחוק את הקריאה של `addAll` בתוך עצם מונה. אבל המחלקות מאד תלויות זו בזו. אם מחר מישחו מגיע ומשנה את `addAll` אנחנו נדפקנו. נרצה למצוא פתרון אחר - עם האצלה! נוסיף שדה של עצם ביןארי לתוך עצם 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(); }

    ...
}

```

```

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 מימינו כאן אותו דבר את `addAll`. הפעם אנחנו משתמשים של `binary-tree` ולא של המונה המורחב. מדובר גם כאן שיש דינמי בידיניג - אבל הפעם הוא הולך לפי סוג האובייקט. מי שלח אוטה? סוג `binary-tree`? מסוג `delegate`? ולכן שנעשה ה `addAll` מסוג `binary-tree` ולכן נשותמש ב-`addAll` הפעם!

כתייה כזו של מחלקה נקרא *pattern design decorated* - קשטי! הוספה של התנהגות או שדות תוך כדי שימירה על שדות והתנהגות מקוריים. הקישוט מותבצע ע"י האצלת כל הפוקודות לאובייקט המקורי בתוספת שדות והתנהגות רצויים.

דוגמה חשובה: אנחנו בפרויקט חשוב ורוצים לרשף שתי מחלקות שונות. למשל נעבד עם

משולשים - יש לנו מחלוקת של משולשים מסתובבים ויש מחלוקת של משולשים עם נקודות. נרצה ליצר משולש מסתובב עם נקודות. מה נעשה? הרי אי אפשר להרחיב שתி מחלוקת!! **כמובן!**

הרחיב להיות *polygon* ולא נעשה *extends* בכלל. נமמש כך - נממש את הממשק *IPolygon* בשניהם, ונוסיף האצלה לכל אחד מהם. נראה כי אם נבקש *timePassed* קורה משזה מוזר - لأن הוא הולך שכותבים *decorated.timePassed*? הרי זה תפקידו לסובב את המשולש. באופן דומה קורה עם *timePassed* בחלוקת השניה. מה קשتن עשויה? קודם כל - תעשה מה שlimido'ו אותן, אני מבטיח לשומר על התנהגות ואז אני ארחיב עליה! ככלומר - אם המשולש יודע להסתובב, שיסתובב. אח"כ נוסיף עליו ונסובב אותו שוב - אני שומר **תכונות מקוריות** ורק **מרחיב אותן**. או לחלופין - אם יש כבר נקודות בתוך המשולש, אנחנו נציג אותן כי זה התוכנה שלו ואני 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 פעמיים נוספת והפכנו אותו למשולש שיודיע להסתובב! מתחילה מפנים החוצה.

כמה סוגים של משולשים נוכל ליצור? כמה שנרצה! אנסוף! נוכל ליצור איך וכמה שנרצה ונוסיף את הקישוטים עליי מלמעלה - ככלומר ניצור *base* וائز נקשר אותו כמה שנרצה! זהו פתרון טוב לכך שלא יכולנו לרשף מחלוקת **ולכן זה הפתרון הטוב**.

נשים לב שאין לנו הביטה שיגיע שם בהכרח משולש - יכול הגיעו כל *polygon*. איך נמנע מכך?
נגיד *interface* חדש *ITriangle* שירחיב את *IPolygon* ולא יוסיף לו כלום - עכשו כל הקשתנים יהיו חייבים לממש *ITriangle*. כל משולש כתעת ימש את *ITriangle* ולא את *IPolygon* ולכן מרובעים למשל לא יוכל להכנס לבפנים. כי הם מסווג *IPolygon*.
בעיה נוספת - יש לנו הרבה שכפול קוד: ניצור *abstract class* *abstract class* לכל המשתפים. אח"כ המחלוקת האחרות ירחיבו אותו ויסכתבו מחדש מה שרציך - בהתאם. כך -

EXAMPLE: TRIANGLES

The diagram shows the implementation of the Decorator pattern for triangles. It includes the `ITriangle` interface, the `TriangleDecorator` abstract class, and two concrete decorator classes: `RotatingTriangle` and `SpottedTriangle`.

```

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();
    }
    ...
}

```

A callout bubble points to the `getArea()` method in the `TriangleDecorator` class with the following note:

נזכיר "TriangleDecorator" כדי לא לאפשר למשתמש ליצור אובייקטים מסוים זה. התוצאה מכך מחייבת האצלה עבור כל מחתודות 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
    }
}

```




סיכום: האצלה לעומת הורשה

יתרונות:

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

חסרונות:

- איבדנו את `a - is` ואת יתרונות הפוליפורמייזם
 - מעט פחות יעיל
 - פחות קריא וקשה ל邏יק
- נשתמש בהורשה כאשר: אנחנו מተכנים היררכיה מחלקות משלנו מתקיימים עקרון `a - is` ועקרון `the class of meaningful subclasses of the hierarchy, even if it is not a child of the base class!`
- אחרת נעדיף האצלה**

2 דפוס שני: Factory – Design – Pattern

דוגמה ראשונה:

מדובר בדף של מפעל - כלומר בכל פעם `pattern` מייצג אובייקטים לפי ביקוש המשמש. נתבונן בדוגמה של משחק "איקס עיגול". כמו בדוגמה כאן. מכרכנו את המשחק ללקוח - הוא חוזר ונמא ס לו שהיוזר לא יכול להכנס את הבחירה בעצמו לשיבו המוקם. `user` משעמס אותו - נרצה `.smartUser`. איך נ ממש? צריך לעמוד 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();
    }
}
```

כינוי שמחזרו את המשחק שלנו
ללקות, והלקות מבקש להוציאף –
שיי סוגים של שחקנים –
BoardPlayer שבודח תא על הלהת
אוון חכם, HumanPlayer –
שambilקש מהמשמש לבוחר תא
על הלהת.

ממושך מתקלחות
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() {...}
}
```

צטריך לבנות מחלקה Game
עבורי כל צורף אפשרי של
שחקנים....

האם אפשר אחרת ?

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

...

נוכל ליצור *Iplayer interface* שם יהיה *Game* כדוגמה בצד ימין
יראה המחלקה לאחר שתובן. זה לא טוב! במבנה של *Game* אנחנו חייבים להגדיר מראש שני
אובייקטים של *players*. כאן בדוגמה אתחלונו זאת מראש – מי יודע מה *user* רוצה ואיך הוא רוצה
שהשחקנים במשחק יהיו? נקבל מה *user* בargs מה הוא רוצה שייהה. ניצור ממשך חדש –
השחקן *supplier* אשר יספק מתקלות כמפורטות מטה, אך "כ" ב-*main* כשניצור נוכל לקבל שני
supplier ובעזרת המתודה *get* נתחייב כי אכן יהיה קל להוציאף כמה סוגים שחקנים שנרצה ללא ש ינווים
דרמטיים בקוד.

FACTORY DESIGN PATTERN – THE

```
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);
}
```

במשך הקורס נלמד
כלים שאפשרו לנו
לשכל את הפתרון הזה

מתוך זה נראה
Factory Design Pattern.
יצירת אובייקט מוביל לדעת מהו
הטיפוס המדויק של האובייקט.
זה פתרון לפי עקרונות
של OOP.



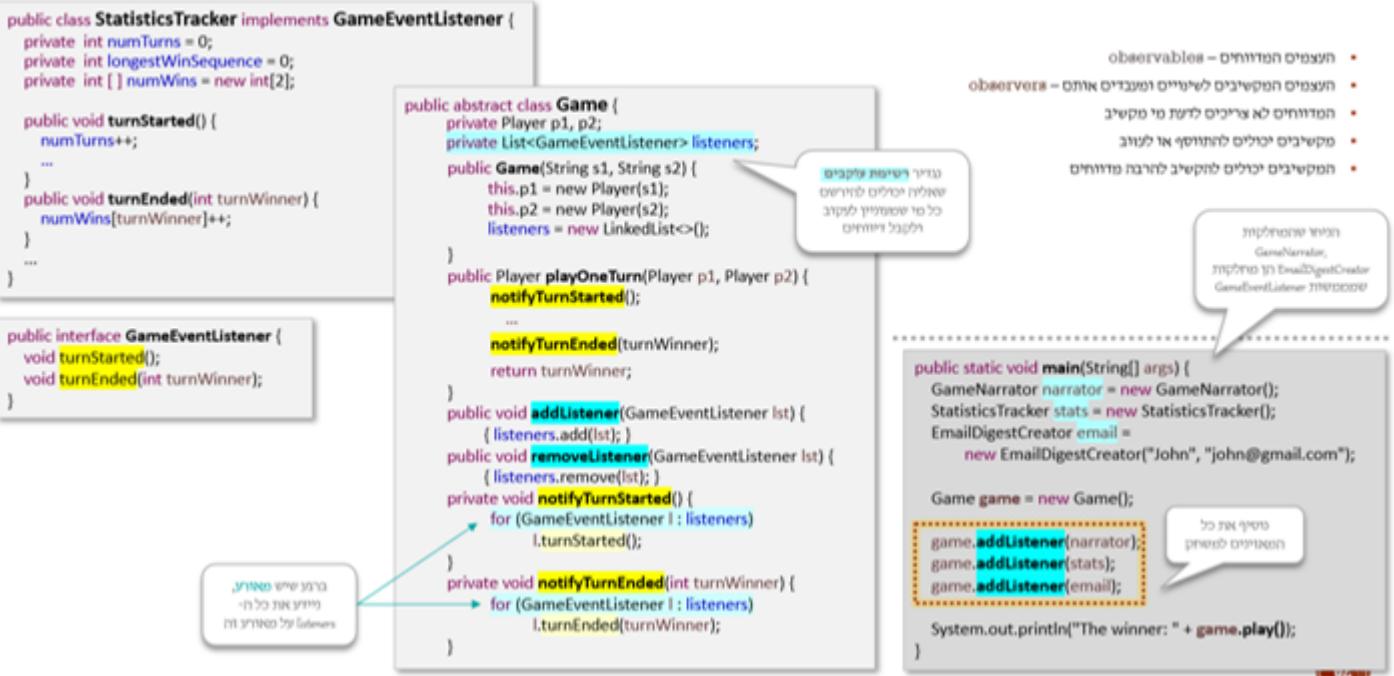
כיתה צו עם supplier היא בדיקת ! כנראה לפועל כך משתמש בשיטה *Factory – design – pattern*.
זו.

דפוס :observer

דוגמא: לכל אחד מאיתנו יש חשבון אינסטגרם עם עוקבים, כל אחד מהעוקבים הוא *observer*. כל אחד שעוקב אחרינו באינסטגרם צריך להיות מסוגל להציגו (לעקב) ולהפוך *observer* או כאשר הוא רוצה להוריד עוקב - שיכל להוריד מאיתנו עוקב וכן כאשר אנחנו מפרסמים *reals* נרצה שכל אחד מהעוקבים יקבל הודעה על כך! נרצה שהעוקבים "ינהלו" את עצם. במצב זה משתמש בדפוס *הןיל*.

זוג מה נוספת: נרצה לשמר סטטיסטיות על משחק - מס' נצחותו לכל שחקן ורץ נצחות מksamלי. איך נממש זאת? *observer* שלנו כאן יהיה *StatisticsTracker*. במחלקה *Game* נוסיף *listeners* של *listner*, כמו כן נממש שם מתודות של הוספה ומחיקת *listner*. בכל רגע שקרה מאורע נקרא לכל *listeners*.

OBSERVER DESIGN PATTERN



דפוס: iterator

- * יש לנו אוסף של אובייקטים ונרצה לעבור עליהם אחד אחרי השני. נרצה לבצע על כל אחד מהם משהו. מה אם נשתמש `ArrayList`? סבבה. אבל מה אם נעשה זאת על `LinkedList`? זה לא יעיל, כיון n שאנו ניגשים ב($O(n)$) לכל איבר לחפש אותו (הרוי המחשב מחפש אותו כי הוא לא רציף בזיכרון 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` שיידע לעבור בצורה ייעילה על מבנה הנתונים (ambil להכיר מראש את מבנה הנתונים שכן נרצה לשמור על `!encapsulation`!).
- * באופן תאורתינו יכולנו לחשוף את המבנה הפנימי של המידע שלנו - לשבור את `encapsulation` וכן לבצע עיל. אבל זה לא טוב! גול חושפת את מבני הנתונים בהם היא משתמשת? ממש לא.
- * לכל ממבנה נתונים נבנה `iterator` שיוכל לעבור על המבנה (תמיד יראה אותו דבר!) כך שנוכל בכל מבנה נתונים להשתמש בו.
- יש ממש שכתבו לנו מראש וניתן להשתמש בו

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

כמו כן, תחזיר האם יש איבר הבא אחרי, וכן `next` תחזיר את האובייקט מהסוג הגברי הבא אחרי במבנה הנתונים. נשים לב שנוכל להוסף לממשק באופן `remove` תאורתינו גם אך לא חובה למשוך אותה כMOVED.

את אותו `interface` נדרש ממש במבנה הנתונים שלנו!
נראה דוגמה למימוש שאותה מחלוקת 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    }

```

איך משתמש בזה בע ת?

```

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 הכמהה וגם קל מאד להחליפ מבני נתונים.

*ניתן להשתמש ב *itertor* גם עבור דברים כמו חישוב האיבר a_n -י של סדרת פיבונאצ'י, נגיד בקונסטרקטור שנשלח איבר כלשהו n_i ונחזיר איבר n בפיבונאצ'י. איך נממש? זה כל עוד יש לאיירם חישיטים בפיבונאצ'י ליציר (הרוי בשבייל לחשב איבר אני צרייך לייצר את $1 - n$ קודמוני). נראה גם שנווכל למשמש די בקלות את *next* בהינתן $a = F(n - 1)$ ו $b = f(n - 2)$ נראה כי

$$\begin{aligned}
intcurrent &= a \\
intsum &= a + b \\
&\quad a = b \\
&\quad b = sum \\
&\quad remaining - \\
&\quad return current
\end{aligned}$$

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

Nested – class

יש עדין ב *itertor* בעיה. לעיתים הוא כן צריך לדעת את המבנה הפנימי לפחות עבור *itertor*. ככלומר שרק הוא ידע, אבל כרגע אי אפשר לחושף אותו כי איןנו חלק מהמחלקה. אכן משתמש ב *itertor* בלי לשימוש על הכמהה. מדובר על מחלוקת בתוך מחלוקת!!!

```

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 נגידרה כמחלקה פריבט גם לא יוכל ליצור אובייקט ממנה.Cut נוכל ליצור את ה- *itertor* בפנים שיכיר את המבנה ולא ישבר עקרון ה- *encapsulation*.
כן היינו רוצים לקרוא לאופן *itertor* כללי מבחוץ - וכן אם נגידר את המחלקה הפנימית כ- *static*, זו מחלקה שモוגדרת פ"ר המחלקה יכולה המקורית שמכילה אותה.
***הערה** - יש ממשק נוסף בשם *Iterable* שהוא אומר שניתן לשימוש ב- *itertor* על מבנה הנתונים.

******* טוב לדעת - הסיבה שモותר לעשות *for – each* כמו *for(Integer – num : collection)* היא שמאחורי הקלעים ממומש *itertor* של המחלקה שנקרה, וכך כן מותר לעשות רק על אובייקטים שהם *iterable*.

הרצאה 8 - *:Cloning&Immutability*

Immutable - לא ניתן לשינוי
Mutable - כן ניתן לשינוי.
ישנם שפות רבות שהינס *Immutable*, כלומר הגדרנו $3 = x$, זהו - אי אפשר לשנות אותו. נגלה עליהם בהמשך וגם מדובר צריכה את זה ומדובר לא אפשר לשנות. נגלה על כך בקורס "שפות תכנות".
***Clone**: יש לי אובייקט? נרצה להכפיל אותו ולקבל אחד זהה כמוו.
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 ומישם מבחן יכול לגשת למידע זהה ולשנות אותו כי שלחנו *reference* אליו **איך נפתר את הבעיה?** נרצה ליצור העתק של הסטודנט על מנת שימושו מבחן לא יוכל לגשת, אבל זה הרבה זכרו שנטפס וכן אנחנו לא יודעים מה הטיפוס המדויק של הסטודנט. כאשר ניצור העתק לא נדע האם זה סטודנט למדעי המחשב או מתמטיקה... לא עובד!

ב- *clone* **מוגדרת מותודה** *class – Object*
המותודה הינה *Protected*
מחזירה *Object*
יצרת עותק שטхи (*shallow – copy*) של כל השדות של האובייקט המועתק המותודה זורקת חריג עבור מחלקה שאינה משתמשת *interface – Cloneable* אם אובייקט דורש העתקה عمוקה (*deepcopy*), יש לדחוס אותה *cloning* - ממשק ריק, מסמן מחלקות שתומכות ב- *Cloneable* זו מותודה ג'אוית שקיימת בתוך *classObject*, החתימה שהיא כך

protected – Object – clone() – throws – CloneNotSupportedException.....

על מנת לאפשר ל-*class* *clone* לגשת הוא חייב למשה את הממשק *cloneable* - זהו ממשק ריק!
לא מתחייבים לכלום, אנחנו רק מקבלים אישור מה*jvm*. הירק כאשר ננסה להפעיל את המתודה על מחלקות שלא ממשות את *cloneable*.

נחוות לדוגמה מוקודם - כתעת אם ניצור סטודנט כלשהו ב-*main* וזו נגידיר $s2 = \text{student}$ – $s1 = s1.clone()$; נקבל שנווט עתקה של *Object clone* – זה נקרא *shallow-copy*, העתקה רדודה! זה לוקח את כל השדות ופשטוט מעטיב את הערכים *asIs* בלי לחשב על ההשלכות. בואו לחשב על ההשלכות... יש לסטודנט שם, משה. אנחנו יודעים כבר שם יש לנו שתי מחרוזות באותו שם הנשמרות באוצרן באתומו מקום! ככלומר קיבלנו שלכל סטודנט יש *id* נפרד, אך בשם המש מביעים אל אותו שם (כי ככה עובד *String immutable*). קיבלנו אובייקט וחציו!!!..... במקורה שלנו זה תקין! כיון שהוא *String immutable* ולא ניתן לשנותו. מסקנה: אם יש לנו שדות פרימיטיביים שהם *reference* זה בסדר לנו. אם אחד ינסה לשנות את שמו הוא קיבל חדש לשם החדש.

איך נגידיר עבור Clone – Student Class? כתעת הוספנו אובייקט בשם כתובות לשדות, נתחיל ב-*shallow clone* רגיל ואח"כ נctrיך לעשות שוב *clone* לאובייקט השני – כתובות, זה נקרא *deep clone*. ייחד עם זאת נctrיך לדרוס את המתודה *clone* אצל *Address*? גם שם *clone* מדוע? יש אובייקט *City*!*City* *clone* לעשות שוב *super clone*, לקרוא *shallow clone* עם *deep clone*! ואח"כ לבצע *City clone* עבור *City*. ואז סיימנו – אין צורך לעשות *clone* ב-*City* כי השדה שלה הוא *immutable*. כתעת שנרצה ליצור סטודנט ב-*main*, נוכל להשתמש במתודה *clone* כמו בדוגמה על מנת להעתיקו.
(הדוגמה עליה דיברתי מצורפת כאן בתמונה)

```
public class Student implements Cloneable {
    private String name;
    private int id;
    private Address address;

    @Override
    public Student clone() throws CloneNotSupportedException {
        Student copy = (Student) super.clone();
        → copy.setAddress(getAddress().clone());
        return copy;
    }
    ...
}
```

על מנת לבצע *deep cloning*

```
public static void main(String[] args) {
    Student s = new Student("Moshe", 1234,
        new Address(new City("Kfar Saba", 50000), "Weitzman");
    try {
        → Student s1 = s.clone();
    } catch(CloneNotSupportedException e) {}
}
```

```
public class Address implements Cloneable {
    private City city;
    private String street;
    public Address clone() throws
        CloneNotSupportedException {
        Address copy = (Address) super.clone();
        → copy.setCity(getCity().clone());
        return copy;
    }
    ...
}
```

על מנת לבצע *deep cloning*

```
public class City implements Cloneable {
    private String name;
    private int numOfCitizens;
    ...
}
```

בBOR מתקיים אין צורך לנגדיר *deep clone* מכיוון שתשודות של כל אובייקט *immutable*, אז מותרת *shallow cloning* ייירוטם מ-*Object*

לסיכום – תמיד נדרס את *clone* של *object* עד שנגיע לאובייקט שהוא *immutable* – ושם לא נדרש לעשות *deep – Clone*.

כעת, נוכל לפטור את הבעייה 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 לבעה הקודמת. בואו נניח שהסטודנט היה *immutable*, זה היה הרבה יותר קל עם `clone` אחד. אם המורה, גם כן...
 יצרנו מחלקה שהיא *ImmutableList* וכאן במחלקה *exam* יצרנו את הרשימה כאובייקט של המחלקה *ImmutableList*. כמובן - רשימה שאינה ניתנת לשינוי. גם היא תצטרך למשמש את השדה שבפנים יהיה רשימה שהיא *private immutable* אבל היא תהיה *cloneable*. אז לא ניתן יהיה לשנות מבחוץ ואם נבנה את המחלקה בצורה נכונה נשמר על *encapsulation*. שוב, נהיה חייבים למשתגרים וסטריטים וכן *remove add* וכן *copy.l.add* וכו'!
 *הערה - במתודת *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<T>(); }
    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<T>();
        for(item : l)
            copy.l.add((T)item.clone());
        return copy;
    }
    ...
}

```

אנחת מוסיפים item לשותק שלו, ולא לנכמיתו

כעת נזכיר, מדוע בכלל בינו את המחלקה? להמנע מזבוז זמן העתקות - וכך בדיקת אנחנו

mbutzim annosof hautekot! *add* anchno mutikim at col hareshima.... chbel ul hazon uvel haizkoron. vlonen nafzel lemkrim - zo talui b'makraha vba'afliktsia shlano! **shni haftorotot shelno groueim.** ma pachot groue? am ani pachot meudchen at hareshima az neshemesh b'sh yisha ha'zo vam anchno hraba meudchenim at hareshima neshemesh **b'fturon hakodim.**

- * מומלץ מאוד (לא חובה ולא עקרונית *oop*) במחלקה *ImmutableList final* את המחלקה (לא ניתן להרחיב אותה - כי הרכהה עולה להרשותם את תכונת *immutability*) וכן את כל השודות *final* - ואם ננסה לשנות את *reference* שלו קיבל שגיאה (זה הרי היה בטוען) - ניגש לשיטה זו רק כאשר לא נרצה כמעט כמעט בכלל לעדכן את הרשימה.

:Immutability-disadvantages

1. **חסרוו ראשוו** - בדוגמא כאן הסיבוכיות תהיה $O(n^2)$ למراتות 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"
→ s -> "aaaaaa"
```

פתרון: נפעיל שיקול דעת, במרקרים כמו כן אנחנו מרצה את `string` להיות *mutable*! לשם כך גאווה מודעת ויצרה `StringBuilder` למרקרים כאלה 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(); //
```

2. **Stack – Immutable**: לא העתקות! נבוד הרבה יותר קשה ויצירתי אך יש קבוצת אנשים ש חשבו והשיבו בזה ויצרו מבני נתונים Immutable | שהוא גם עיל:

IMMUTABLE STACK

```

public final class EmptyStack
    implements ImmutableStack {
    ...
}

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 {
    ...
    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; }
}

```

בשימוש כאן כל הפעולות ב($O(1)$). מה קורה כאן בעצם? המחסניות נבנות אחת על השנייה כך שהמחסניות הריקות נמצאות למיטה. מיהו *prev*? המחסנית הקודמת. בכל מחסנית יש "איבר אחד" פלוס האיברים שנמצאים מתחתיה. זה בדיק רshima מקושרת... כל אחד זכר את זה שלפנוי עד המחסנית הריקה. מחסנית שמיוצגת ע"י רshima מקושרת. מה קורה במחיקה? אנחנו רק נחזיר את הכתובת של הקודם, וה*garbage collector* ינקה אותו כי אף אחד לא מצביע עליו!

(מומלץ לראות את הkkלטה של מרינה לנושא זה!)

כל השודות הינם *private* השdots נוצרים בתוך המחלקה ולא יוצאים החוצה איברי המחסנית הינם מטיפוס *Integer* מתודות שימוש את התוכן של המחסנית הינו פרטיות נקראות אך ורך חלק מיצירת עצם או העתק שלו

תכונת Singleton: הגבלת מס' המופעים של המחלקה למופע יחיד (ניתן גם למס' קבוע של מופעים) מודיע זה חשוב? בדוגמה שראינו לעלה נרצה להגביל את מס' הפעמים שאפשר ליצור מחלקה ריקה - וכן נשתמש בתוכונה זו. מופע זה ישמר כשדה סטטי (יחידי) של המחלקה ויוחזר ע"י מתודה סטטית של המחלקה. ככלומר נגידר את הבניי להיות *private*!*create* ויצור מתודה *empty*. **למשל היכרה תראה כך**

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

ואז נממש מתודה שתהיה אחראית ליצור את המופע היחיד זה

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

סיכום של תבניות עיצוב

1. מוטיבציה - *Observer/listner*:

נרצה נשנה משהו, כל המקומות שקוראים לו ישנו גם כן מalgo. יש קשר בין רבים. בכל פעם שיתבצע בו שינוי הוא יבצע *notify* לכל האחרים שיתעדכו גם כן. למשל, כאשר כדור עבר מהתוות מסויימת נבצע *notify* שיעדכו.

למעשה התבנית מאפשרת לאובייקט אחד להודיע לאובייקטים אחרים (הצופים בו) על שינויים בalgo. 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*. אפשר ל-*Observers* להציג את עצם מהרישה. מכל פונקציה(*notify*) ששולחת עדכון לכל ה-*Observers* הרשומים. **הצופים - *Observers*:** אובייקטים שמחברים ל-*Subject*. מקבלים הודעה אוטומטית כאשר יש שינוי ב-*Subject*. מישמים משק מסויים, לדוגמה, *update()*.

2. *Decorator* (האצללה):
מדובר בקשתן. משתמש כאשר נרצה "לkeletal" את האובייקט בתוכנות נוספות. למשל נרצה להוסיף צבע לכדור, או לוסף אותו וצדומה. נוסיף מחלקות שיקשו אותו מסביב. נבע מכך שלא ניתן לרשף מושני מחלקות ולכן הפתרון הוא להשתמש בתבנית עיצוב - נוסיף שדה של *ball*. הקשתן מאפשר ליצור מחלוקת בסיס עם תוכנות והתנהגוויות שיוגדרו מראש, ולהוסיף תוכנות רצויות (למשל לצבעו אותו) באמצעות הקשתן. בנסיבות הקוד זה נראה כאילו ממש אנחנו מקשטים את האובייקט הפנימי ביותר החוצה. הקשתן מרחיב את התנהגות בזמן הרצה, לא בזמן קומpileציה, لكن יותר גמיש ודינמי.

3. *Factory*:
מדובר בדף של מפעל - כולם בכל פעם *pattern* מייצג אובייקטים לפי ביקוש המשתמש. נפריד בין יצירת המופיע לבין שימושו במופע. איך הפקטוריה בנו? יש משק או מחלוקת אב אבסטרקטית *factory* עם מתודה *create* ומחלקות שיממשו אותה / ירושות אותה ומהוות מפעלים של אובייקטים ספציפיים.
מתי נשתמש בתבנית זו? כאשר מתודה בחלוקת מסוימת מבקשת ליצור מופיע של אובייקט ספציפי, שאינו יודיע מראה מיהו, אלא זה תלוי בבחירה המשתמש (למשל בתוך משחק אקס עיגול) ולכן נשתמש במפעל. המתודה מקבל בתוך פרמטר את המפעל המתאים וтвор את האובייקט כך:

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

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

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

5. *Singelton*:
הגבלת מספר המופעים של אובייקט למופיע יחיד (או כמה, מס' קבוע). איך עושים זאת? נגיד בחלוקת שדה סטטי בשם *instance* בו את המופיע של המחלוקת. ההפוך את בניין המחלוקת לפרטי, כך נמנע יצירת אובייקטים מחוץ למחלוקת, ניצור מתודה סטטית שתחזיר את המופיע, היא תבודוק אם *instance = null*, אם כן היא תקרה לבניין ותחזיר את המופיע המשמש. אחרת, הוא כבר קיים והוא תחזיר את השדה הסטטי (المופיע) :*Comperator* .6

נרצה להשוות בין אובייקטים. התבנית *comparable* (מمشק בג'אווה) משמש לסידור של אובייקטים לפי תחין מסוים. מחלקות שימשו את המחלקה *compareTo* שמשתתפת בה המתודה *compareTo* שמבצעת השוואת מסוים בין האובייקט הנוכחי לאובייקט אחר מאותו הסוג.

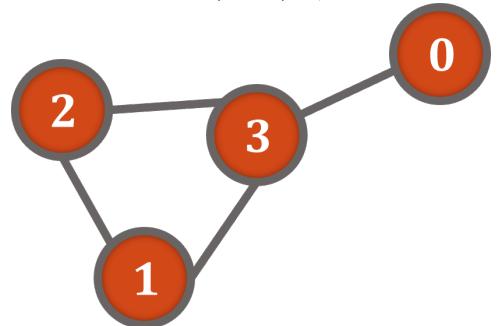
התבנית השנייה הוא אינטראפיס *comparator* של *compertor* שמשמש לסדר מותאם אישית של אובייקטים. הממשק מכיל את המתודה *compare* אותה יש למשש שמשווה בין שני אובייקטים מאותו סוג. נרצה ליצור מס' מחלקות כאלה על מנת למיין אובייקטים בדרכים שונות. (מצחיר את הרעיון הכללי במבוא של מיוון)

7. Iterator: בשימוש בו כאשר נרצה לעבור על אוסף כלשהו מבלי שהוא יודע אם 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 *oop* עם גרפים.

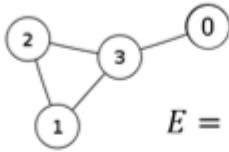
תזכורת: גרף $G = (V, E)$ מורכב מרכיב קודקודים וקבוצת קשתות $V \times E \subseteq V$. למשל -



קיים כבר בג'אווה: *interface*

```
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() ; }
```

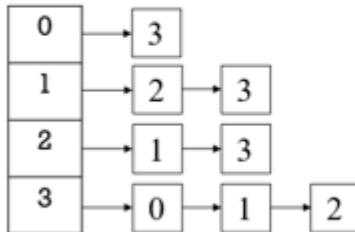
דרך למימוש גרף: אפשרות אחת היא כמו בmbao עם רישימה מקוורת, דרך אחרת היא באמצעות מטריצה כמו שופיע ערך מימין. אם יש מפגש של שתי הקודקודים נסמן זאת עם T ואחרת אין קלומר F . למען הפשטות נבודעם מחלוקת *Edge* שתציג צלע (קשת), כאשר יש לה *left* ו-*right*.



$$V = \{0,1,2,3\}$$

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

We would implement class for each representation



	0	1	2	3
0	false	false	false	true
1	false	false	true	true
2	true	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();
    }

    ...
}
```

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++)
            adj[i] = null;
    }

    public int numberOfVertices() {
        return nVertices;
    }

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

    ...
}
```

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

Let's add parent class



Adjacency List

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

***הערה:** על הנייר היה נראה טוב ליציג גרף עם טבלת גיבוב כך שחיפוש הוצאה ומחיקה יהיה $O(1)$, אבל איך נמשפ 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 $O(n) < O(degree)$.

נרחיב את הדיון וניצור קלאס אבסטרקטי ובכל גראף חדש אנחנו נמשפ את שלושת המתודות **שפטה: 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.

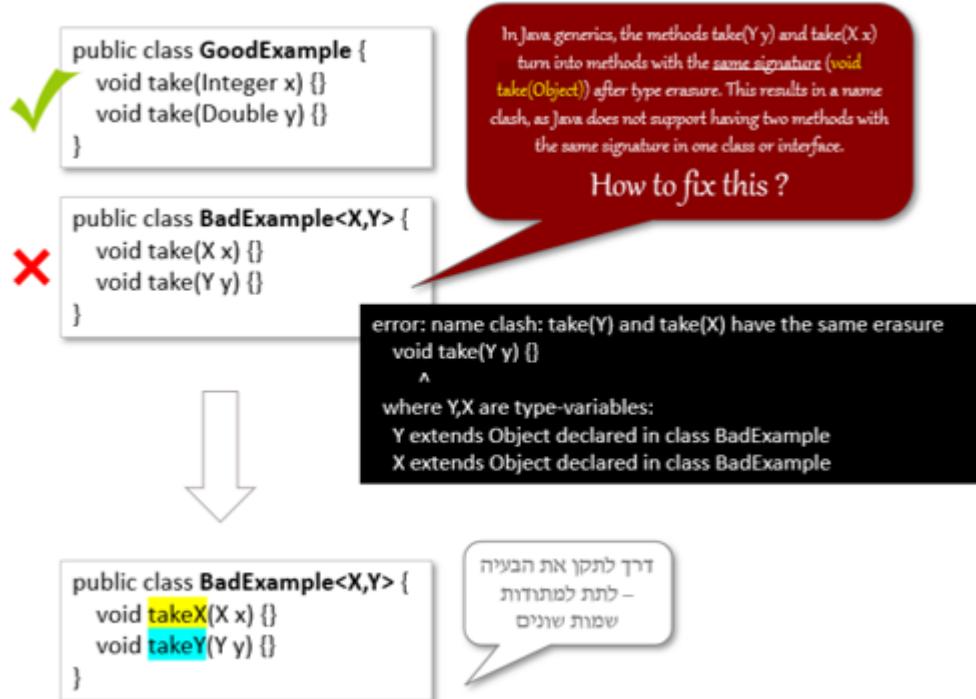
```
public class EvenDegreesAlgorithm extends GraphAlgorithm
```

```
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
}
```

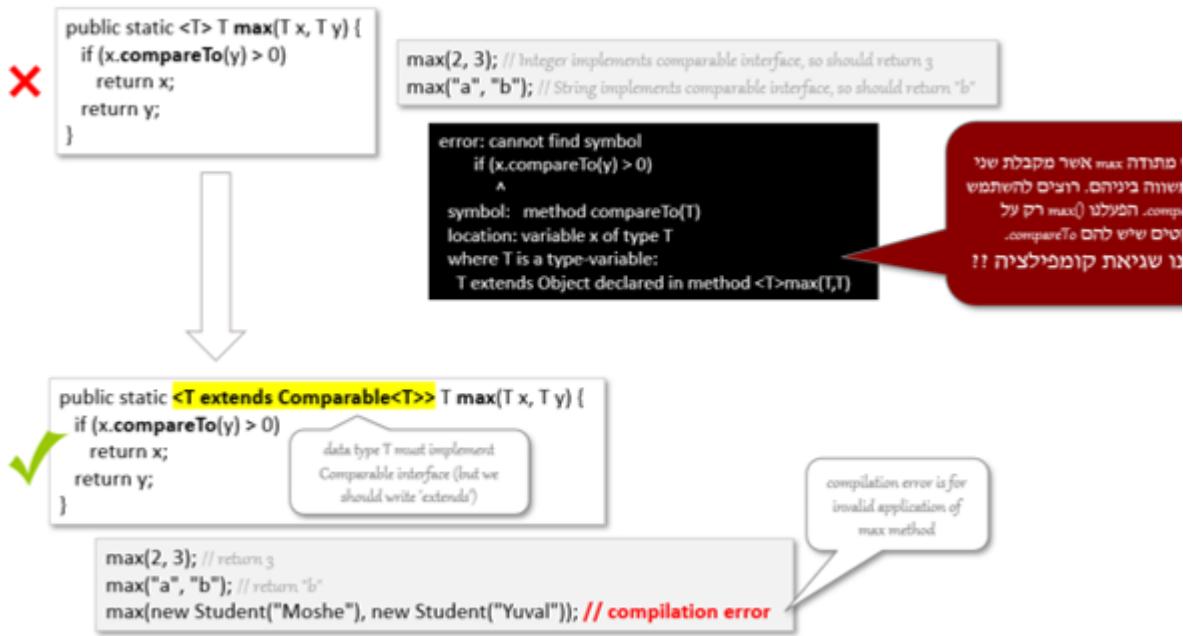
הרצאה 10: תכונות מתקדמות ב-OOP: טרייקים שלדים בגאואה

1. על פניהם הניר המתודת למטה נראה תקינה, אותו שם אך טיפוס שונה. זה בסדר. כת נסתכל על *Bad - example*. באשר הוא גנרי נקלט שגיאה. מה השגיאה? הקומפיאר יגיד שיש להם אותו *erasure* - בזמן קומפליציה המתודות הופכות שתיהן ל-*take(Object)* ולכן יש לנו שתי מתודות עם אותו שם שמקבלות אותו דבר... וזה שגיאה.



דרך התמודדות: לשנות את השם של המתודות. לא קיימת דרך אחרת להתמודדות עם מצב זה. בכל הklassים הגנריים המתודות בסוף מתורגם ל-*Object*.

דוגמה חדשה: מתודה גנרטית. בדרך לкопמל את הדוגמה מטה נקלט שגיאת קומפליציה. מה? הקומפיאר לא מוצא את המתודה *compareTo*, מדוע? *Object* הוא לא ממש את ממשק *comparable* (*Comparable* לא מושך *int* ושאר הפרמייטיבים), כמו שאמרנו קודם גנריים מתורגם ל-*object* בזמן קומפליציה. מה הפתרון? נוסיף לחתימה הרחבה של ממשק *comparable*. נבדוק בכל פעם למשה כאשר נעשה קראיה *max*. מה הבעה? למרות שחוובה לעשות מימוש *comparable* *student* יתכן שמחלקה כמו *max* לא עשתה ואז בקריאה שלה נקלט שגיאת קומפליציה.



תכונה נוספת על generics: כתעת נשלח אל *comparator - cmp* *max*, כך לכארורה יוכל להשוות ללא *compareTo*. הקראיה הראשונה חוקית. אנחנו רוצים להראות שהוא מוסיף מגניב - הגדרנו בתוך הקראיה של המתודה שימוש של מה?! עשינו *new* ל*interface*?! מה שקרה כאן הוא שני דברים - **יצרנו קלאס אונוניים!** אין דרך לדעת מה השם של האובייקט (מאתה ה

- comparator
- < String >

 מוגדר), מדובר באיזשהו קלאס שה*jvm* מגדיר, הוא מימוש את שם מזרך כלשהו), ולאחר מכן ממש את המתודה היחידה שיש בתחום המחלקה האונונית הזו - כיון שהוא ממש את *comparator*, חייבים ממש 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. קיורי דרך. ניתן לעשות → בג'אווה. מה מטרתו? נראה דוגמה:

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

זו כתיבה וכי מקוצרת שיש בג'אווה כאשר אנחנו מגדירים קלאס אונוניים. מה אמרנו כאן לקומפילר? כאשר אתה תפעיל את המתודה *max*, אתה יודע שאתה צריך לקלל שני שמות וכן כלשהו. ניתן לשולח *comparator* מקוצר כבר בתחום החתימה. ניתן באופן דומה לעשות

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

מה תפקיד החז' החל מהחץ → מתחילה המימוש של המתודה/ ממشك.
למעשה, את המתודה הפשוטה

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

ניתן לכתוב כ

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

לשימם לב זה אפילו בלי נקודה פסיק, מודיע? התוכונה מגיעה במקור מפיפיתון.

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

***מתודה** `arr.forEach((i) -> system.out.println(i))`: על `arrayList` ניתן להפעיל על אובייקט מסווג זה

כasher היא תעבור על כל האיברים ברשימה / מערך ותעשה מה שנדרש ממנו מימיין.

נראה כי הערות הבוחרות היכלות הינס **מחלקה אונונימית!** זהו המימוש של המחלקה `run`

שמופיעה לעיל. איך יודעים זאת? **לפי החתימות!** אם היה עוד חתימות בתוך `stringFunction` לא

יכולים 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);
    }
}
```

למעוד Java בנתה מאחוריו הקלאסים
שתי `class`, `anonymous classes`, `עובר כל`
גדרת של מטודה וזה. בנוסף
להגדרת מחלקות, `java`, בנתה שני
אובייקטים, אחד מכל מחלקה.
חויפות `exclaim` ו- `ask` מצביעות
לאובייקטים אלו.

Hello!
Hello?

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

```

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 `for`, המחלקה שתיווצר תהיה זהה לכל הולאות. לעומת זאת, כאשר מחקים את רעיון `for` ומשכפלים קוד 3 פעמים יוצרים 3 מחלקות שונות.

*** חשוב לזכור** - כאשר כותבים `getClass` נוצר אובייקט.
3. עבר מחלקות אבסטרקטיות: נרצה להבנות Aiיזהו `abstract lambda` מחלקה. הדוגמה מימין כן תעבור - עם מחלקה אונומית.

```

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();   Hi
    }
}

```

4. reflection: יש לנו מחלקה ואנחנו מעוניינים לקבל מידע עליה. ראיינו שאנו יכולים לעשות זאת עם `getClass`. כתת נתבונן ונבון מהדוגממה מטה - מסתבר שיש פקודה `DeclaredFields` שמחזיר `collection` של כל השדות, لكن ניתן להפעיל עליו `length` ולדעת כמה שדות יש. גם יש שמחזיר `Field` מסתבר. מסתבר שבאופן דומה יש פקודה שמחזירה רשימה של המחלקות - מחלקה בשם `Field`. וגם כאן יש מחלקה בשם `Method`. יש פקודה בשם `getSuperClass` - היא מחזירה `null` על `object` בלבד, ועל כל מחלקה אחרת היא תחזיר את המחלקות שירשיהם מהם. **כאר נדפיס את כל המתודות כMOVEN שנדפיס גם את המתודות של כל מי שאנו יורשים מהם** - ואנו תמיד יורשים מ-`Object`! זה הורס `encapsulation` שכן כל ההגנות נשברות ויכולים לשנות תוכנות מסוימות. **native** - מתודות שלא ממושכות ב-`java` וлокחים אותם מ-`c` או `+c`. לא נרchip כרגע ולא זה חשוב. (לא לבחן)

```

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 ובטע שלא נדבר על איך שהוא בנווי. ניתן להפעיל אותו באופן יידי מתווך התוכנית - אם כי זה לא מומלץ! איך קוראים לו? `system.gc()` קורא לו והוא יעשה מה שצדיק לעשות.

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

```

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

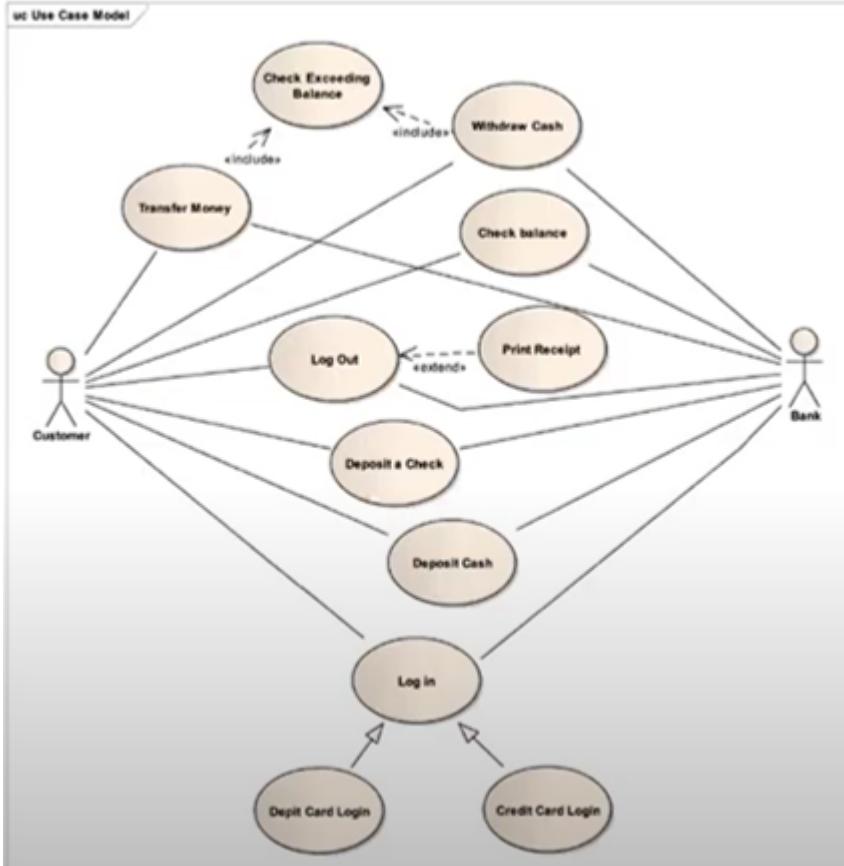
```

7. *UML*: הדרך שלנו לכתוב דרכים שייעזרו 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. נctrיך להבין אילו אינטראקציות קיימות. למשל במערכת של בנק: כאשר אני מושך כסף - מי מעורב בזה? חשבו הבנק, הבנק, מערכת הפריטה. כאשר אנחנו ניגשים לפרויקט, עוד לפני שניגשים לתכנת, פועלמים בשלבים הבאים:

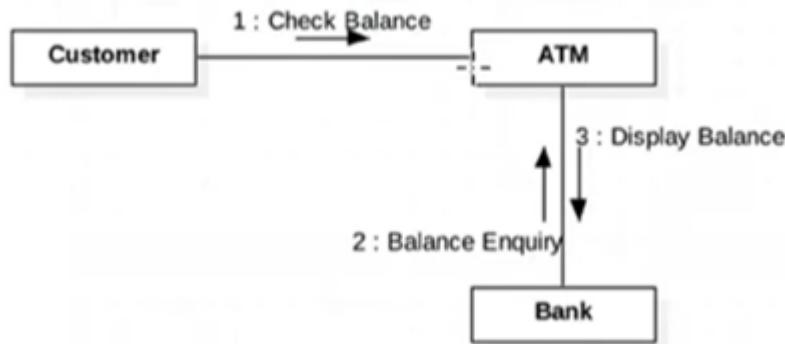
- חשובים מה הדרישות שיהיו מהמערכת, לרוב הדרישות יגיעו מחלוקת שלנו. אם פרויקט קיים ומספרים - אילו שיפורים ל��ח רוצה ממנו לבצע.
- לפני לתכנן: מיהם השחקנים במשחק, אילו מחלוקות צריך עבורם, איזה פעולות אנחנו רוצים שהמערכת תבצע?

- תכנן: לסדר את המערכת לפי מחלוקות והורשה, ולעשות כמה *uml* שונים. *UML* מתחלק לשניים. יש יכולו שגדירים את המבנה (כמו זה שראינו) ויש יכולו שעוברים על התנהלות. מדובר בסה"כ בתיאור ויזואלי של המערכת.

1. *Diagramma של התנחות*: *use – case*: הדיאגרמה לוקחת אובייקטים שבמערכת, ומסבירה עם חצים מה הקשר בינם מבחינה התנחותית. דוגמה -



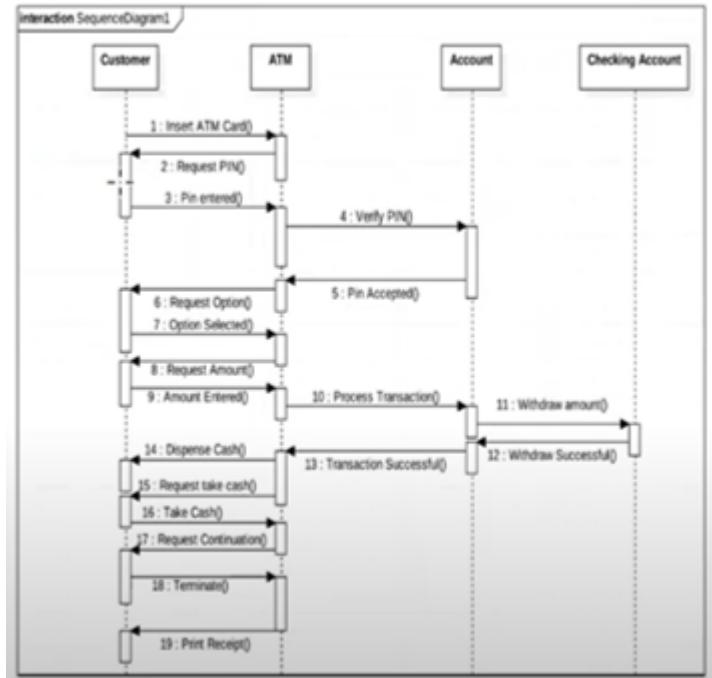
2. **דיאגרמה נוספת נראית** *diagram cummunication* שתסביר את הקשר בין המחלקות השונות בהפעלת פעולה. למשל בדוגמה מטה - הלוקה מפעיל את הפעולה, שם זה הולך *ATM*, ואח"כ לבנק. ההבדל בין זו לקודם הוא שמדובר רקנו את החיבור בין העצמים, וכעת אנחנו רואים את הסדר שכורה בזמן הפעולות.



3. **state – diagrams**: כמו במודלים בתיכון, יש אוטומט כלשהו ובו מתואר הפרויקט. כמובן אם יש לולאה עצמית אז נשארים בתוך המצב עד לשינוי, וכן ניתן לעבר במצב בהתאם לדרישות הפרויקט שלו. כאשר כל אחד מהמצבים לא מצליח, הוא ילק למצב כשלון שם 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.

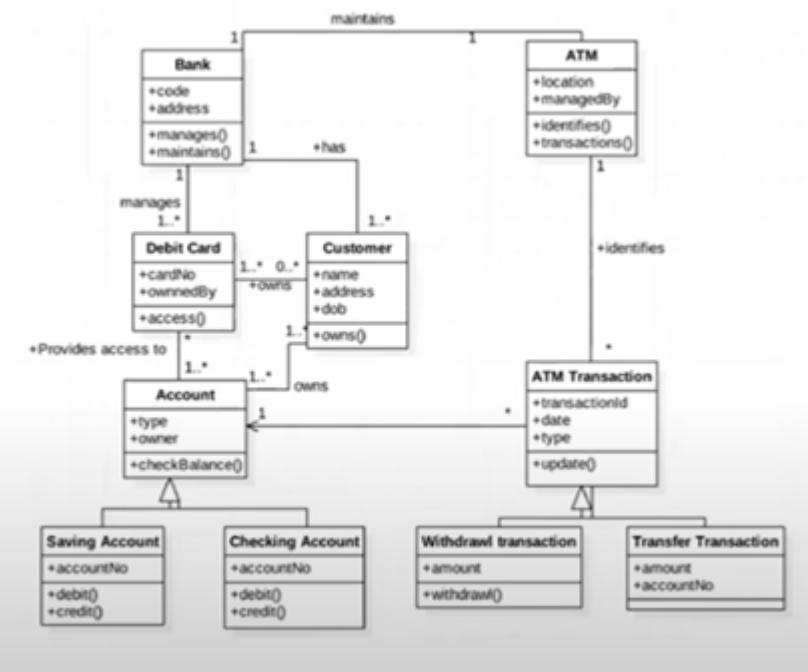
5. **Sequence diagram** : דיאגרמה מאוד חשובה לפני צביקה. מציקה גם את התקשרות בין הפעולות. ניתן לראות ממש את החצים של התהליך בזמנים זמינים, כמה זמן יתבצע הפעולה, איזה פעולות יתבצעו תוך כדי? המלבנים מייצגים את הזמן גם כן. אחת הדיאגרמות החשובות.



המסקנה מכל הדיאגרמות היא שבכל פרויקט נשתמש בcaçãoו, 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/ אינטראפיקס וכו' בהםם. ח'ז מדבר על ירושה, ח'ז מוקוקן מדבר על ממשק, המספרים 1 מתיאר כמה יכול להיות, למשל לכל בנק יש ATM אחד, אבל לכל בנק יכולים להיות הרבה לקוחות *customers*. הרבה מסומנים ב*.לפעמים כתוב על היחס שהוא - מה היחס, לא חובה. מה זה המועוין? מעוין ריק - מצין הכליה, מעוין מלא מייצג "קשר חזק". למשל - שיעור לא חייב להיות תלוי בסטודנט - יתכן שיור ללא סטודנטים. אבל יש קשרים כמו תלמיד, ותלמיד למדעי המחשב. לא יתכן תלמיד למדעי המחשב שאינו תלמיד. ח'ז מצין הורשה.



סימנים של *class – diagram*

* - *# , public = + ,protected* -

הלוּם - הכליה משמעותית, יש תלות של מחלוקת בחלוקת אחרת (למשל המחלוקת היא שדה שלה)

קו מקוקו - משתמשים במופע של המחלוקת השניה באחת המתוודות, אך אין תלות
8. *TDD: test – driven – devolpment*: עשית דיאגרמה. כתבתת אחהלה קוד. הרצת כמה פעמים.
מצאת באג. תיקנת בbag. כתבתת טסט, עבד. סבבה. כאן זה הפוך - קודם נכתב טסט. נכשל
בו. כעת המטריה לפתח את הקוד שלא נכשל בו. המטריה הפוכה: כעת אנחנו מנסים להתכוון
לטסט, ולא סתם על עיור ואז להכין טסט. בכל פעם אנחנו נבצע טסט, אם נüber אותו ותקדם
לטסט הבא. אנחנו בכל פעם נריץ את הטסט הנוכחי ואת כל הקודמים - וזה מאוד חשוב, שכן
אם טיפולנו בbag אחד יכולנו ליצור אחד אחר. לכן זה יבטיח שזה לא יקרה.