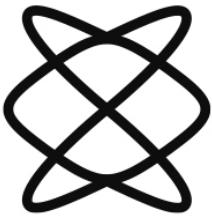


**החוג למדעי המחשב (0368)
תוכנה 1 (2157)
(קורס ארוכה)**

מרצה: לנה דנקין
מתרגלת: אלה גולדשטיין
תשפ"ג, סמסטר א' (2022-2023)

מסכם: רועי מען



**The Raymond and
Beverly Sackler Faculty
of Exact Sciences
Tel Aviv University**



פרק 1 - מבוא

4.....
עבודה בסיסית ב-Java

פרק 2 - מחלקות

12
17
22
מחלקות
מנשיים
טיפוסים גנריים ומחלקות פנימיות

פרק 3 - ירושא

32
ירושא I
36
ירושא II
42
חריגים

פרק 4 – נושאים מתקדמיים

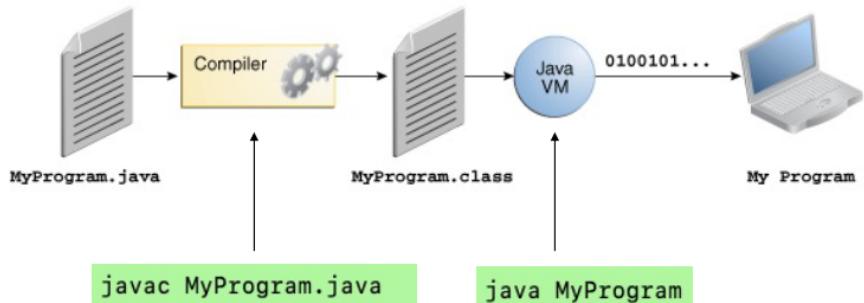
48
תכונות פונקציונלי וארמיים
58
ירושא III (מתקדמת)



1 – מבוא ל-Java

עבודה בסיסית ב-Java

שלום עולם



אם מתחילה מקובץ עם סימן זה – בו כתבים את הקוד בשפת java. הקוד עבר שני שלבים.

1. השלב הראשון הוא **שלב הקומפלול** (בשלב זה מתבצעות בדיקות syntax) בו נוצר קובץ

בסימות class – זו שפת ביןים, קובץ ב-

Bytecode (הוראות ריצה ב"מחשב כללי")

בשלב השני מתבצעת **אינטראפטציה**

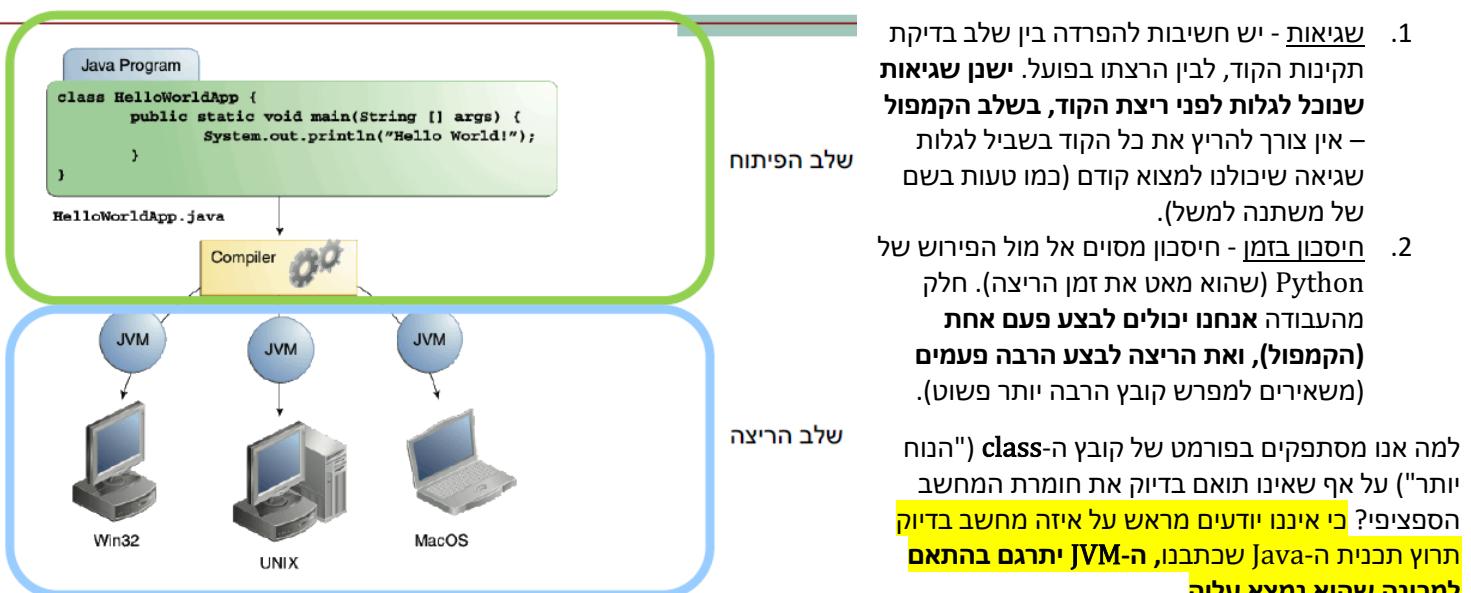
להריצה על הסביבה המתאימה שבה אנו

רצים שהתוכנה תרוץ, **אמצעות ה-JVM**.

לכל מערכת הפעלה נדרש JVM אחר.

Java היא שפה אוניברסלית, ולכן בשלב הקומפלול הוא שמייצר פלט אחיד – קובץ class. כדי להריץ אותו צריך לדעת על איזו מערכת הפעלה אנחנו נמצא, וシリוה JVM מתאים מותקן. **מדוע לנו צרכי שלב נוסף בתהליך, למה זה לא עובד כמו ב-**

?python



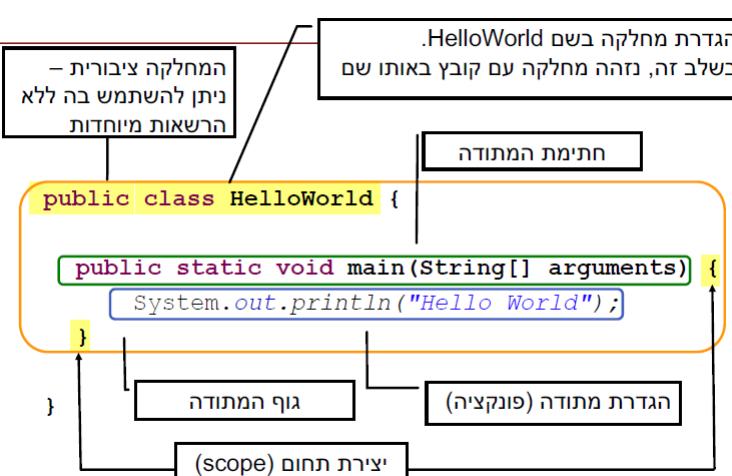
הערות:

- לרוב שמו-class יהיה זהה לשם הקובץ, קובץ יוכל בדרך כלל מחלוקת אחת (אפשר גם לשיטים יותר מחלוקת אחת).
- JRE – יודע להריץ classים של Java, לרוב מותקן כבר על המחשב. כולל את ה-JVM, הכל שיודיע על קחת bytecode

ולהמיר אותו לשפת המבונה.

- **כל קוד חייב להופיע בתוך class כלשהו**, בתוך פונקציה כלשהיא. הוא לא סתם "מරחף".

כאשר אנו מרכיבים class כלשהו, ה-JVM מחפש את המתודה main עם החתימה הידועה שלה, ומרים אותה. זה הדבר היחיד שרצ. יכול להיות class ללא פונקציית main – כאשר אנו מיצרים ספריה, ולא תכנית ראשית שנרצה להריץ.



עובדת עצמית 1

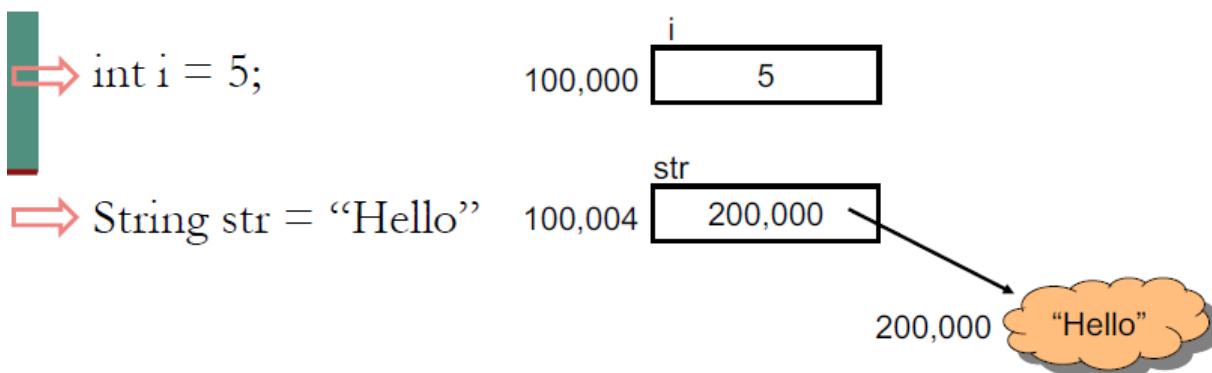
1. בתרנו את התכנית הבסיסית **HelloWorld**:
 - א. ב-Java כל הקוד נכתב במחלקות. לא ניתן לכתוב פונקציות/להגדיר משתנים בתוך קבצים מחוץ למחלקה.
 - ב. הפונקציה `main` מקבלת מערך של מחרוזות ולא מחייב שום ערך.
 - כ. אין שימושות תחריבית לרווחים וירידות שורה, פתיחת סגירת בלוק נעשית באמצעות סוגרים מסולסים.
 - ד. בסוף כל פקודה علينا לשים נקודת פסיק (').
2. **הגדרכנו משתנים:**
 - א. הכרנו 8 סוגי טיפוסים בשפה.
 - ב. **כל טיפוס יש גודל ידוע מראש.**
 - כ. **מחרוזות נכתבות בין מרכזות "" בעוד תוויים נכתבים בין גושים ".**
 - ד. אפשר לחבר ביןתו למספר שלם, כאשר התו מומך לערך שלו לפי קידוד ASCII. אם נרצה להציג את התוצאה בתורתו אז נבצע המרה (char).
 - е. מספרים שלמים מיוצגים על ידי byte, short, long, int.
 - f. מספרים עשרוניים מיוצגים על ידי float, double.
 - g. ערכים בוליאניים על ידי Boolean, **תו על ידי character** (שונה ממחרוזת, ובפרט ממחרוזת עםתו אחד. מחרוזת היא טיפוס אחר בשפה).
3. **ראינו פעולות מתקדמות על טיפוסים פרימיטיביים:**
 - א. כאשר מדובר ב-`and` (&&) יש לבדוק את שני ערכי הביטויים אם הראשון T, כיון שהוא לא מספיק.
 - ב. כאשר מדובר ב-`or` (||) אין צורך לבדוק את שני ערכי הביטויים אם הראשון T, זה מספיק כדי שככל הביטוי יהיה T.
 - כ. האופרטור ++ מגדיל ב-1 את ערך המספר שעליו הוא מופעל וגם מוחזיר את המספר עצמו אם הוא באז ימין להשמה. **כאשר נכתב ++ קודם מתבצעת השמה, ורק אז ++ לערך של 1num=2num**. לעומת זאת אם נכתוב `1num++=2num` ולאחר מכן נקבע שמי המשתנים יהיה זהה.

טיפוסי הפניה ומערכות

פרט ל-8 הטיפוסים שראינו עד כה, **כל שאר הטיפוסים** בשפה אינם **פרימיטיביים**. הספריה התקנית של Java מכילה יותר מ-3000 טיפוסים. מערכות ומחרוזות אינם טיפוסים יסודים, אולם נדרש להם כבר בשיעורים הקרובים ולבן נדון בקצרה בטיפוסי הפניה.

הפניות:

טיפוס הפניה (reference type) – משתנה מטיפוס שאינו יסודי נקרא הפניה (התיחסות/מצבי/פונטרא). ביצירת משתנה מטיפוס יסודי אנו יוצרים **מקום בזיכרון בגודל ידוע מראש**, שיכל להכיל ערך מטיפוס מסוים. למשל עבורizo נקבע מקום שם ישב התוכן עצמו של המספר. ביצירת משתנה הפניה אנו יוצרים **מקום בזיכרון, שיכל להכיל בתוכת של מקום אחר בזיכרון**, שם נמצא **תוכן בלהשו**. למשל עבור String אנו נקבע מקום שבו **הערך הוא פשוט בתוכת בזיכרון, של מקום אחר שבו יושב תוכן המחרוזת**.



המשתנה `str` נקרא הפניה, בעוד התוכן שעליו הוא מצביע נקרא עצם (object). אזור הזיכרון שבו נוצרים עצמים שונים מאשר הזיכרון שבו נוצרים משתנים מקומיים, והוא מכונה Heap (זיכרון ערימה).

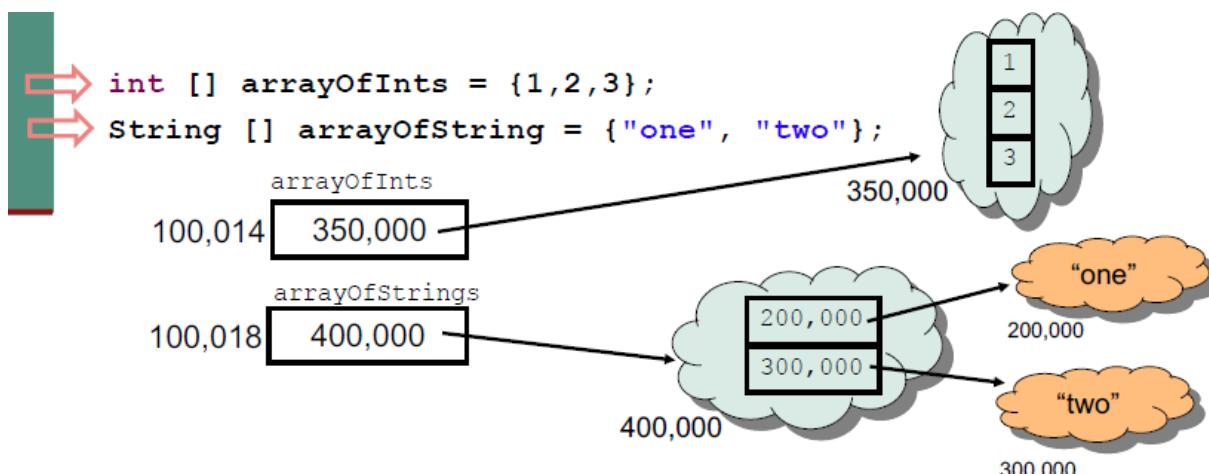
נקודות חשובות:

- **פעולות על הפניה** – השמה למשתנה הפניה שמה ערך חדש במשתנה הפניה, ללא קשר לעצם המוצב. **בהתמה 2=a** התוכן של 2 נרשם בתוכן של 1, בולמר בעט ב-1 נמצאת הכתובת של האובייקט, שגם ב-2 נמצאת הכתובת שלו. **1=a כתעת יצבע** לאותה כתובות בזיכרון שאליה מצביע 2.a.
- **ערך Null** – ניתן ליצור משתנה הפניה ללא אתחולו. כמו ביצירת משתנה פרימיטיבי ערכו יהיה זבל – **אי אפשר להשתמש בערך זה**, ולא ניתן יהיה לגשת אליו. **ערך זה אינו null**. **ניתן להשים למשתנה את הערך null** (כמו None ב-python). כך ניתן יהיה לגשת אליו בהמשך כדי לבדוק אם אותו(str==null).
- **שרשור מחרוזת** – כאשר נבצע שרשור באמצעות + קודם כל נוצרת מחרוזת חדשה.
- **פניה לעצם המוצב** – האופרטור נקודת (.). אפשר גישה לעצם המוצב. בשאנו פונקציה לדוגמה str.toUpperCase, נוצרת מחרוזת חדשה בכתובות חדשה עם המחרוזות **באותיות גדולות**, ונקלט חזרה את הכתובת של המחרוזת החדשה.

מערכות:

מערכת מייצג סדרת משתנים מאותה הטיפוס. תאים במערכת יושבים בדרכ"כ בראץ' בזיכרון (Java) רצתה על מכונה וירטואלית, זה לא מבוטח במאה אחז') בר' שגישה סדרתית אליהם עשויה להיות עיליה (להבדיל מרישימה מקוشرת, לחוב נניח שם רציפים בזיכרון, יותר ויותר לקרוא מהזיכרון). **מערך הוא טיפוס הפניה, הוא אינו טיפוס פרימיטיבי**. התוכן של המשתנה הוא בכתובת בזיכרון שם יושבים ערבי המערכת.

כדי לציין שהמשתנה הוא מטיפוס מערך ניעזר בסוגרים מרובעים. נשים לב שבמערך של String, יהיה לנו מערך של כתובות של המחרוזות בזיכרון (כיוון שכבה זה עם טיפוס הפניה, **הערכים במערך הם הפניות לעצמים הנמצאים במקומות אחרים** בזיכרון). לעומת זאת, במערך של int יהיה לנו מערך של **המספרים עצם** בזיכרון.

נקודות חשובות:

- **יצירת עצם מטיפוס מערך** – ניתן להפריד בין יצירת הפניה (המשתנה arrayOfInts לדוגמה – נוצר עם ערך זבל), והאתחול שלה (יצירת עצם המערך בזיכרון – לשם בר' יש להשתמש באופרטור new).ibri המערך מאותחלים אוטומטית לפי הטיפוס שלהם: הפניה מאותחלת ל-null, מספרים יאותחלו ל-0, Boolean יאותחל ל-false וכו'.
- **בינתן לשאול מערך לאורכו** – זה מאפיין פנימי אשר ניתן לגשת אליו ישירות, מדובר בשדה length. זאת בשונה מ-String.length – שם מדובר בפונקציה שמחזירה את אורך המחרוזת.
- **הפניות ואופרטור השוואה** – כאשר == מופעל על משתני הפניה, הוא משווה את הפניות (הכתובות) ולא את העצים שנמצאים בכתובות הללו. בולמר, גם אם מדובר באותו ערכם במערך, לא בהכרח מדובר באותה הפניה, אלו יכולים להיות מקרים בכתובות שונות בזיכרון.
- **שיטוף** – אם שתי הפניות מצביעות באותו עצם, העצם הוא משותף לשתיهن. **כל אחת מההפניות יכולה לשנות את העצם המשותף** המוצב בזיכרון בלתי תלויה.

הפרוקציה של מערכים ומחרוזות – מכיוון שמחוזות ומערכות הם טיפוסים מאוד שכיחים ושימושיים בשפה, הם קיבלו שתי תכונות שאינן אף טיפוס אחר בשפה:

1. **פטור מ-new:** אפשר ליצור מחרוזות ישירות באמצעות מירכאות ("Hello"), ומערך באמצעות סוגרים מסולסים ({1,2,3}).
2. **הפניות ואופרטורים:** על משתנה מטיפוס הפניה לרוב אפשר לבצע רק השמה, השוואה או גישה לעצם. על מערך ניתן לבצע גם גישה לאייר ([]), על מחרוזת ניתן לבצע גם שרשור (+).

מבci בקרה

תנאים:

- **opertor if/else** – ברגיל. אין elif כמו ב-python, צריך לרשום if ואז else.
- **opertor if/else syntax** – קצת שונה מ-python. ההשומות יכולות להיות גם תוצאות של חישובים והפעולות של פונקציות.
- **נשים לב כי חיבת להבצע השמה למשנה כלשהו על מנת להפעיל את האופרטור זהה.**

ב Python

```
x = 1 if y == 0 else 8
```

ב Java

```
int x = y == 0 ? 1 : 8
```

```
String day = args[0];
int numLetters = switch (day) {
    case "MONDAY", "FRIDAY",
    "SUNDAY" -> 6;
    case "TUESDAY" -> 7;
    case "THURSDAY", "SATURDAY" -> 8;
    case "WEDNESDAY" -> 9;
    default -> {
        System.err.println("Illegal day " +
                           day);
        yield 0;
    }
};
```

ריבוי תנאים (**switch-case**) – תחביר מיוחד לריבוי תנאים. ניתן לסייע משפט switch לאחר ההטאה הראשונה על ידי המילה break (אם אין break ממשיך ל- case שנמצא מתחתי בaczera אוטומטית). מקובל למקם default (לא חובה) באפשרות אחרת, במננה לכל הערכים שלא הופיעו ב-case משליהם.

החל מ-15 Java אפשר לבצע **switch גם עבור ביטוי**, כמו להציג `{...} int num = switch(day) { ... };`. אנו מבצעים השמה ישירה לתוך ערך, כאשר ההשמה מבצעת case ומחייבת ערך באמצעות yield. בתחביר מתקדם יותר אפילו אפשר להשתמש בחיצים!

- ריבוי תנאים (**switch-case**) – תחביר מיוחד לריבוי תנאים. ניתן לסייע משפט switch לאחר ההטאה הראשונה על ידי המילה break (אם אין break ממשיך ל- case שנמצא מתחתי בaczera אוטומטית). מקובל למקם default (לא חובה) באפשרות אחרת, במננה לכל הערכים שלא הופיעו ב-case משליהם.
- **החל מ-15 Java** אפשר לבצע **switch גם עבור ביטוי**, כמו להציג `{...} int num = switch(day) { ... };`. אנו מבצעים השמה ישירה לתוך ערך, כאשר ההשמה מבצעת case ומחייבת ערך באמצעות yield. בתחביר מתקדם יותר אפילו אפשר להשתמש בחיצים!

ולולאות:

- **foreach** – מזכיר את לולאת **for** של python עם "in". **foreach (double d : arr)**. בתחביר זה הקומפיילר מייצרת את העבודה עם משתנה העזר בaczera אוטומטית מאחורי הקלעים. **לולאה זו יש פחות כוח – יותר קשה לגשת ולבצע שינוי** ב مكان `[i]`. המשתנה `d` הוא משתנה מקומי, ואם נבצע בו שינוי זה לא ישפיע על המערך. עם זאת, **אם פ' הוא טיפוס הפניה, אנו נוכל לגשת באמצעות כתובות בזיכרון ולשנות שם את הערך עצמו.**

שירותי מחלקה והעמסה

```
<modifiers> <type> <method-name> ( <paramlist> ) {
    <statements>
}
```

שירותי מחלקה: בתוך class יש פונקציות. פונקציה זו (שירות מחלקה) מוכחתת על ידי מילת המפתח static. התחביר של הגדרות שירות:

נקודות חשובות:

- **modifiers** – אם 0 או יותר מילוט מפתח מופרדות ברוחחים (למשל public static) **type** – מציין את טיפוס הערך שהשרות מחזיר **void** – מציין שהשרות אינו מחזיר ערך **paramlist** – רשימת הפרמטרים הפורמליים, מופרדים בפסיק, כל אחד מורכב מטיפוס הפרמטר ושםו
- **משתנים** – גוף השירות מכל הוצאות על משתנים מקומיים, נקרים גם משתנים זמינים/מחסנית/אוטומטיים. הם מתקיימים אך ורק ב-scope של הפונקציה. הגדרת משתנה זמני צריכה להקדים את השימוש בו. **חיבים לאותל או לשיטם ערך מפורש** במשתנה לפני השימוש בו.
- **זימון מתודת** – קרייה לשירות תופיע בתור פקודת ע"י ציון שמו וסוגרים, עם או בלי ארגומנטים. אין חובה לשמור ערך שחוזר מפונקציה, אך אם לא שומרים אותו הוא הולך לאיבוד. **אם אנחנו צריכים את ערך החזרה, עדיף לא לקרוא לפונקציה פעמיים (זה סתם חישוב כפול וגם אולי יכול לגרום בעיות), אלא לשומר אותו ולבזר איתו.**
- **שם מלא (qualified name)** – אם אנו רצינים לקרוא לשירות מתוך מחלקה אחרת, יש להשתמש בשמו המלא של השירות, שכולל את שם המחלקה שבה הוגדר ואחריו נקודה.

העמסה: ב-python אין דרך לקרוא לשתי פונקציות באותו שם. ב-Java זה אפשרי, אם החתימה של הפונקציות שונה: כאמור ו/או במספר הארגומנטים, לא כולל הערך המוחזר. **הגדרת שתי פונקציות באותו שם ובאותה מחלקה** מכונה העמסה.



שלוש סיבות להעמסה:

1. **נכחות** – אם לא הייתה אפשרות לממש שתי פונקציות עם אותו השם, היינו נאלצים להמציא שם נפרד עבור כל מתודה, שמות מלאכותיים לעיטים ולא נוחים. בעורת מנגנון ההעמסה נובל להגדיר זאת בנוחות. בחלק מהמקרים, אם משלימים על הנוחות הזאת באיזו בהירות: לא ברור איזו מהפונקציות תופעל. **הקומפיילר** מנסה **למצוא את הגרסה המתאימה ביותר** עבור כל קריאה לפונקציה על פי טיפוסי הארגומנטים, בעזרת casting שימוש במה שפוחות מידע.

למשל, איזו מהפונקציות תופעל במקרה הבאים:

- `max(long, long)`
`max(1L , 1L) ; //`
- `max(double, double)`
`max(1.0 , 1.0) ; //`
- `max(long, double)`
`max(1 , 1.0) ; //`
- `max(double, long)`
`max(1.0 , 1L) ; //`

2. **ערבי בירית מחדל לארוגמנטים – אין פרמטרים דיפולטיים ב-Java.** ניתן להשתמש כאן בהעמסה כדי לאALTER פתרון.

פונקציה עם פרמטר אחד, תקרא לפונקציה שנייה עם שני פרמטרים ותעביר ערך דיפולטי.

```
public static int func(int x) {  
    return func(x, 1);  
}  
  
public static int func(int x, int y) {  
    return x + y;  
}
```

3. **תאיימות לאחר –** נניח כי במערכת כלשהו כבר קיימת פונקציה המבוצעת חישוב כלשהו. בשלב זה לא ניתן להחליף את חתימת הפונקציה כיון שיש הרבה הקוד שמשתמשים בפונקציה. על כן, **במקום להחליף את חתימת הפונקציה, נוסיף פונקציה חדשה בתור גרסה עם העמסה (פרמטרים שונים).** משתמשי הפונקציה המקורית לא נפגעים, וממשתמשים חדשים יכולים לבחור לאיזו גרסה של הפונקציה לקרוא. **כדי לשמר על עקביות שתי הגרסאות, נמשש את הגרסה החדשה בעזרת קריאה לגרסה החדש – ונמנע שכפול קוד.**

העמסת מספר **בלשוח של ארגומנטים** – נניח שברצוננו לכתוב פונקציה שמחזירה את מוצרן הארגומנטים שקיבלה. המספר אינו ידוע, רעיון אפשרי הוא להעיבר את הארגומנטים כמערך (הלוגיקה סובכת את איברי המערך ומחלקת במספר האיברים). פתרנו את שכפול הקוד, אך יש כאן הבדה על הלוקוח שבדרש ליצור מערך. **קיים תחביר שימושי לקבל מספר לא ידוע של ארגומנטים (vararg). ניעזר בשלוש נקודות (...)** שיוצר את המערך מאחוריו הקלים.

משתנים:

- **משתני מחלוקת (גלווליים)** – מוגדרים בתוך גוף המחלוקת אך מחוץ לגוף של מתודה כלשהו, ויסומנו על ידי `static`. הם שונים ממשתנים מקומיים בכמה מאפיינים:
 - **תחום היברот** – מוכרים בכל הקוד ולא בפונקציה מסוימת. תלוי בבראות (...), `public`, `private`.
 - **משך חיים** – אותו עותק של משתנה נוצר בזמן טיענת הקוד לדיבורו, ובאשר קיים כל עוד המחלוקת בשימוש.
 - **אתחול** – מאותחים בעת יצירתם, אם לא הוגדרו להם ערכים **יאוחלו לערכי בירית מחדל**.
 - **הקיצת זיכרון – הזיכרון המוקצה להם הוא ב-Heap** ולא ב-Stack.
- **final** – ניתן לקבע ערך של משתנה ע"י שימוש ב-`final` (ב-C# const). **למשתנה זהה ניתן לבצע השמה פעמי אחת בלבד.** כל השמה נוספת המשתנה תגרור שגיאת קומPILEZA.
 - נשים לב כי אם ביצעו: `[3] arr = new int[] arr`, לא יוכל לשנות את תוכן הערך של `arr` (הכתובת שאליה הוא מצביע בזיכרון). **אמנם, ניתן לשנות את הערכים של המערך עצמו (מה שקרה בתוך הכתובת),** בתור אובייקט. נוכל לבצע `1 = [0].arr`. **ה-final הוא על המשתנה `arr` בלבד שמכיל כתובות, ולא ניתן לשנות אותו לנוכח כתובות אחרות.**

תכנות פרוצדורלי (מדריך JAD)

אתחול משתנים:

- שפת ג'אווה אינה מתירה לשימוש בתוכן של משתנה אם היא אינה בטוחה שנקבע לו ערך, בולם, שבוצעה השמה של ערך לערך המשתנה. ניסון לשימוש במשתנה לפני שנקבע לו ערך גורמת לשגיאת קומפיילציה.
- אפשר לאתחול משתנים מיד כאשר **מגדירים** אותם, אבל מותר להשאיר אותם ללא ערך ולקבוע את הערך בהמשך. כל זמן שהערך נקבע לפני השימוש הראשון, השימוש במשתנה תקין.

אופרטורים ארכיטמטיים:

- האופרטור ++ והאופרטור -- משלבים חיבור (חיסור) והשמדה. כאשר מפעלים אותם על משתנה, הם מגדילים או מקטינים את ערכו ב-1. לבב אחד מהאופרטורים הללו שתי גרסאות. כאשר הם **מופיעים לפני המשתנה**, הם מחדירים את הערך החדש שלו, ובאשר הם **מופיעים אחרי המשתנה**, הם מחדירים את הערך הישן שלו.

אופרטורים לוגיים:

- האופרטורים הלוגיים השימושיים ביותר הם & & (וגם) ו- | (או). אלה אופרטורים ביןרים עם התנהגות "קצרה". כאשר מפעלים אותם על שני ביטויים השפה מחשבת את ערכו של הביטוי הראשון. אם הביטוי הראשון קובע את הפלט של האופרטור, למשל אם הערך של הביטוי הראשון בהפעלה של && הוא **false**, אז השפה אינה מחשבת כלל את הערך של הביטוי השני, מכיוון שבBOR שהתוצאה תהיה **false**.

מחרוזות, מערכים, לולאות, שגיאות (תרגול 2)

מחרוזות:

- מחרוזת היא אובייקט המחזיק אוסף של תווים. פונקציות נפוצות: `charAt(0)`, `toUpperCase()`, `length()`. נשים לב כי פונקציות אלו **מחזירות בתובת חדשה בזיכרון של מחרוזת התוצאה**.
- כדי להמיר מחרוזות למספרים ניעזר בפונקציות פר טיפוס: `Integer.parseInt`, `Double.parseDouble` פרטיפוסים וכו'.
- נשים לב כי **מחרוזת + מספר = מחרוזת. לעומת זאת, two + number = number + two** מומר אוטומטית לערך ה-ASCII שלו.

מערכות:

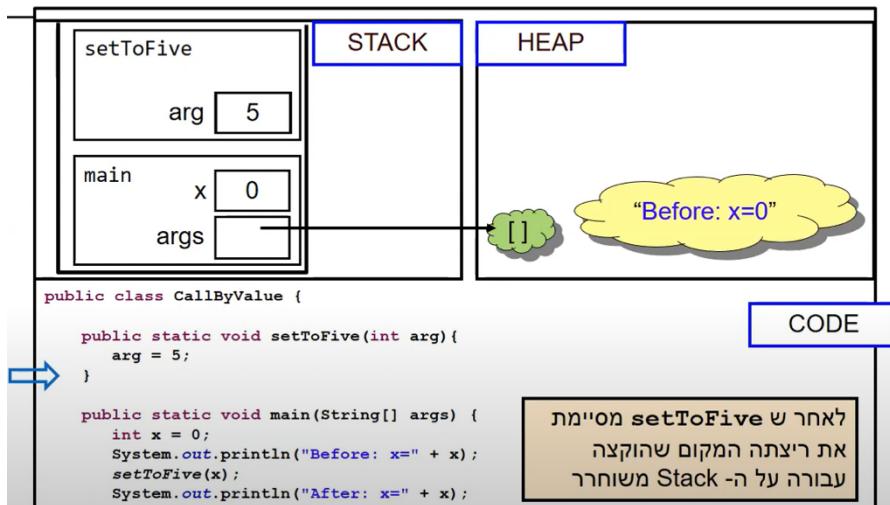
- מערך הוא מבנה נתונים בגודל קבוע מראש אשר שומר מספר איברים **מאותו הטיפוס**.
- אפשר ליצור מערך חדש שיאותחל לאפסים אוטומטית: `[8] int odds = new int[]`, או לשים בו ערכים ישירות: `{1,3,5,7,9} int odds = {1,3,5,7,9}`.
- במחלקה `Arrays` יש הרבה פעולות שימושיות: `copy`, `sort`, `binarySearch`, `fill` ועוד.
- העתקת מערך – באופן נאייבי נבצע `arr2 = arr` אך זה יגרום לשני המשתנים להצביע לאותו מערך בזיכרון, לא מתבצעת העתקה של האיברים עצמם. **בשביל זה אפשר להשתמש בפונקציה copyOf**.
- אפשר לבנות מערכים דו-מימדיים: `char board = new char[3][3]` ואז בולאליה לאתחול כל תא במערך שיצביע `board[i]=new char[3]`.

מודל הזיכרון של Java

העברת ארגומנטים – כאשר מתבצעת קריאה לפונקציה, ערכי הארגומנטים נקשרים לפרמטרים הפורמליים של הפונקציה לפי הסדר, ומתבצעת השמה לפני ביצוע גוף הפונקציה. בהעברת ערך לפונקציה, **הערך מועתק לפרמטר הפורמלי**. צורה זו של העברת פרמטרים נקראת **call by value**. כאשר הארגומנט המועבר הוא הפניה, העברת הפרמטר **מעתיקה את ההפניה** (בלומר גם reference מעבר reference).

ישנם 3 אזורים בזיכרון של התבנית:

- קוד התבנית: מה אנחנו מביצעים, איפה אנחנו נמצאים.
- Heap (ערימה) – משתנים גלובליים ועצמיים (**טיפוסים לא פרימיטיביים**) – אינם תלוי במתודת הנוכחית שרצה. כל טיפוס הפניה, נוצר כאן.
- Stack (מחסנית) – משתנים **מקומיים (פרימיטיביים)** וארוגמנטיים – כל מתודה משתמש באזור מסוים של המחסנית.

העברת פרמטרים:

משתנים בשם זהה הנמצאים בתחום עופף או גלובלים (נראה זו בהמשך בהקשר constructor). מתחודה מכירה רק משתני מחסנית הנמצאים באזורי שהוקצה לה על המחסנית (frame).

3. העברת טיפוס הפניה (מערך) – הfonקציה (arr[]) – הfonקציה setToZero(int arr) מבצעת השמה של מערך חדש בגודל 3, למשתנה arr. גם פונקציה זו לא עשויה כלום, רק בצורה אחרת. הfonקציה **עורכת את המשתנה זם המקומי שיבעיר לכתובת חדשה, אבל arr שנמצא ב-main לא מושפע מזה בכל**.

הערה: בשיטת העברת value על ידי למתודה לשנות את הארגומנט שקיבלה, מכיוון שהיא מקבלת עותק. אז איך יכולה המתודה להשפיע על ערכים במתודה שקרה לא? ע"י ערך מוחזר, או גישה למשתנים או עצמים שהוקטו ב-Heap.

מתודות שימושת תומנת היזכרון נקראות בהקשרים מסוימים Mutators/Transformers. עדכון משתנה גלובלי בתוך פונקציה – **משתנה גלובלי חייב להיות ב-Heap, כדי שוכול לגשת אליו מכל מקום. אילו הוא היה ב-Stack לא יוכל לגשת אליו מכל פונקציה.** הפונקציה increment מבעcit עדכון למשתנה global שנמצא ב-Heap.

הערות:

- איך נכתב פונקציה שצריכה להחזיר יותר מערך אחד – הfonקציה תחזיר מערך. ואם הfonקציה צריכה להחזיר נתונים מסוימים שונים, היא תקבל ארגומנטים הפניות לעצמים שהוקטו ע"י הקורא לפונקציה (למשל הפניות למערכים), ותמלא אותם בערכים ממשמעותיים.

אם נרצה שהfonקציה לא תחזיר ערך – המחלוקת Optional עוזרת לモות את ההתנגדות הרצiosa, נראה בהמשך.

מחלקות בטיפוס נתוניםשירותי מופע:

```

public class MyDate {
    public int day;
    public int month;
    public int year;
}

```

שירותי מופע הן פונקציות אשר מופעלות על מופע מסויים של המחלקה, באמצעות התחביר הבא: object.methodName(args). זאת בשונה משירות מחלקה (static): ClassName.methodName(args). שמות של מחלקות לרוב מתחילים באות גדולה, ושמות של משתנים מתחילים באות קטנה.

נתבון במחלקה MyDate לייצוג תאריכים, המכילה את השדות day, month, year. הם אינם מוגדרים static שכן בכל מופע עתידי של עצם מהמחלקה יופיעו השדות האלה (שדות מופע).

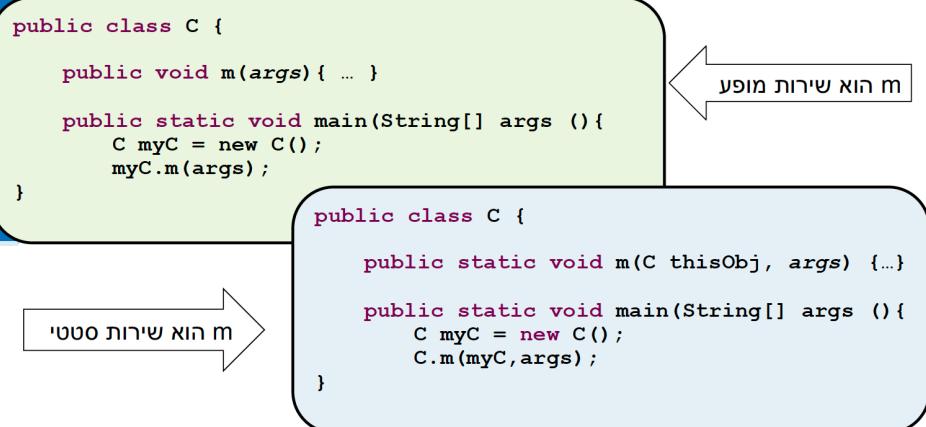
כאשר ה-JVM טוען לזכור את המחלקה, איפה בזיכרון נמצאים השדות? הם עוד לא נוצרו! הם יוצרים רק כאשרLKוק יוצר מופע מהמחלקה, הם לא נוצרים פעם אחת עבור המחלקה (אילו היו static, היו נוצרים פעם אחת ללא צורך ליצור מופע מטיפוס MyDate, וגם עבור 5 מופעים השדות הם מופיעים במקום אחד בזיכרון).

איך יודעים ש-MyDate הוא טיפוס הפניה? הוא לא אחד מ-8 הטיפוסים הפרימיטיביים! لكن חיבורים לשימוש ב-new ליצירת מופע חדש מהטיפוס.

פונקציה סטטית לא יכולה לשמש כפונקציית מופע ולהיפך.

נשים לב לדברים הבאים:

- מה השירות מספק? – האם התאריך הוא תמיד תקין? איזו פונקציונליות המחלקה מספקת לשימוש? המחלקה היא מודול. אחריותו של מודול להמשך את הלוגיקה הנלווה ליצוג תאריכים ולביצוע פעולות עליהם.
- **בראות פרטית** – שדות private לא ניתנים לעדכון ישיר מחוץ למחלקה. גם הפונקציה `isLegal` היא private כיון שהיא לשימוש פנימי של הלוגיקה במחלקה, ואין שום צורך שהלקה יכיר אותה.
- **שירותי מופע** – שימוש בפונקציות גלובליות (סטטיות) הוא מסורבל, לעומת כל פונקציה כזו צריך להעביר את המופיע כארוגמנט. לכן ניעזר **שירותי מופע (פונקציות לא סטטיות)** – פונקציות המשווות למופע מסוים של המחלקה. מאחרו הקלעים, הקומפיילר מייצר משתנה בשם `this` ומעביר אותו לפונקציה (לא נרשום אותו בראשית הפרמטרים, ניתן לשימוש בו במשמעות שירותי מופע, אבל לא חובה להשתמש בו. במקרה שלא משתמשים בו, הפונקציה תבודק האם **למשל d1 הוא משתנה מקומי, ואחרת האם מדווח בשדה של המחלקה – (this.d1 –).**



ניתן לראות בכך syntactic sugar, לדמיין שירות מופע באילו היה שירות מחלקה המקובל עצם מטיפול המחלקה בתוך ארגומנט.

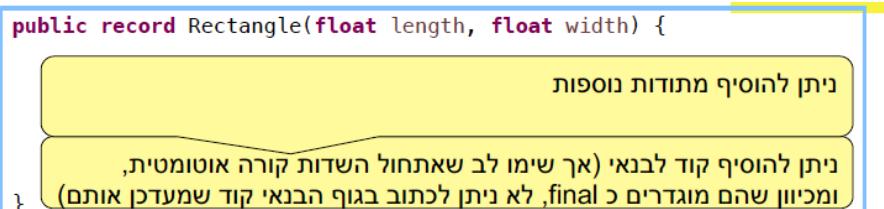
שירותי מופע מספקים תוכנה – נספח פרט לסוכר התחבירי –
המאפשרת החלפת השימוש בזמן – הריצהopolymorphism. תיאור שירותים – מופע בסוכר תחבירי הוא פשוט ו淺易!

:constructors

מיד לאחר השימוש ב-new קיבל עצמו במצב לא עקי, עד לביצוע השמת התאריכים הוא מייצג תאריך לא חוקי (00/00/00). לשם כך ניעזר במבנה – פונקציית אתחול הנקראת על ידי שימוש ב-new. שמה בשם המחלקה שהיא מתחילה, וחთימה אינה כוללת ערך מוחזר. המוטיבציה מהמרכזית היא יצירת עצם שהוא עקי עם השימוש המועד שלו.

נשים לב:

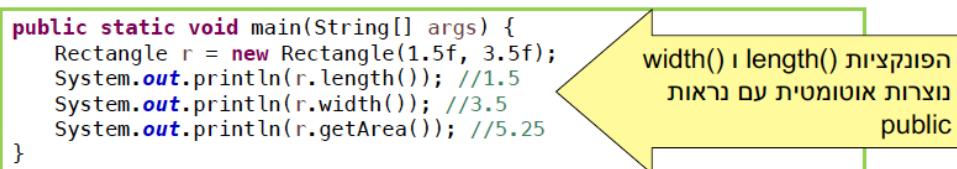
- לא ניתן לאתחל שדה **נסוף בטור הבנאי** – בדיקת שם שלא ניתן לאתחל ערך של משתנה שלא הוגדר (מה הטיפוס שלו?).
- לא ניתן לוותר על השימוש ב-this **בטור הבנאי** – השמה של day=day היא חסורת ממשמעות, הבנאי לא עושה כלום.
- **בנאי ברירת מחדל (default constructor)** – רק כאשר לא הוגדר אף בנאי אחר במחלקה, נוצר בנאי ברירת מחדל, שמאפשר יצרה של אובייקט מהמחלקה ללא שימוש ארוגמנטים.
- **שדות מחלקה שלא אותחו בנאי – מאותחים אוטומטית לערכיהם הדיפולטיים** של כל טיפוס. לצורך העניין אם נגדיר במחלקה `public static int counter` מאותחן דיפולטי ל-0.

:records

סוכר תחבירי חדש ב-Java שמאפשר להגדיר מחלקה שמייצגת plain data carrier, כלומר מאגדת יחד כמה פרטי מידע שאמורים לעבור ממוקם למקום, כאשר המידע לא אמרור להשתנות לאורכו חי המחלקה.

:model

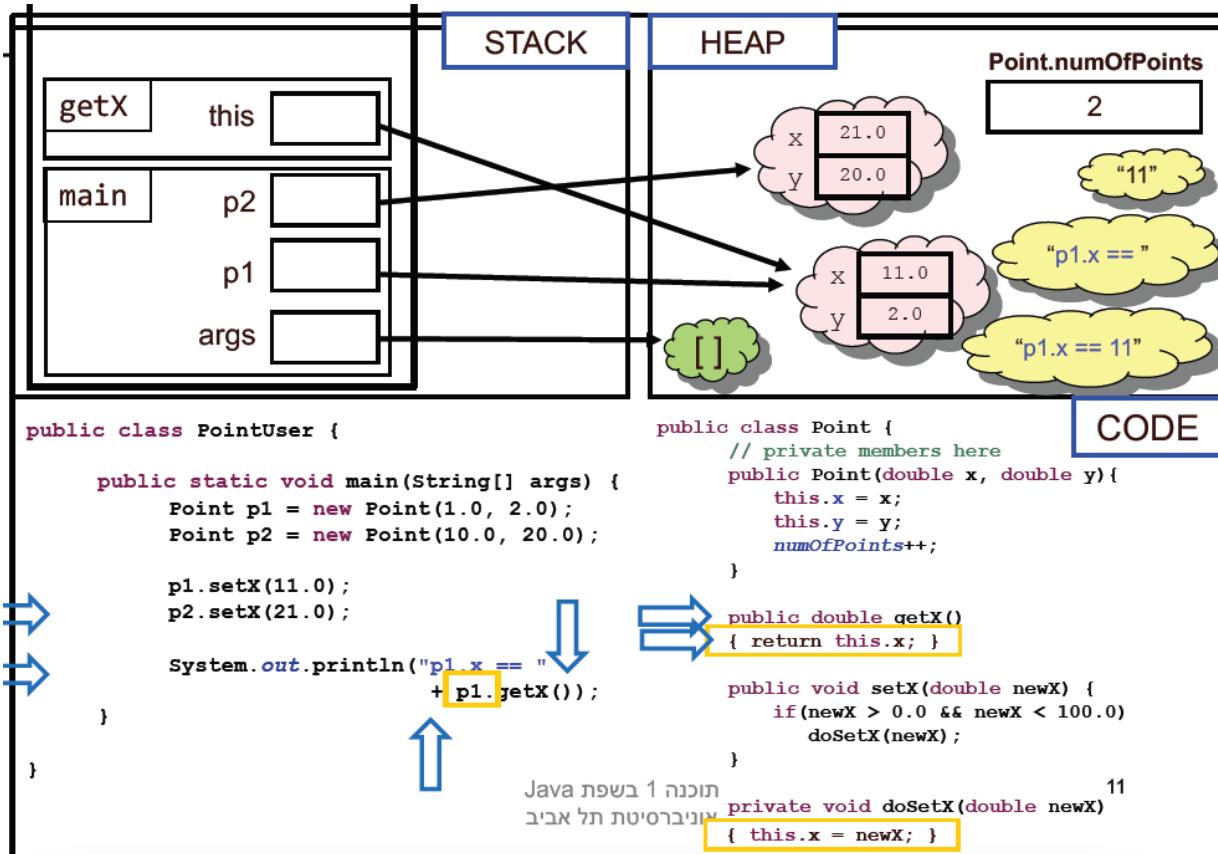
נתונה לנו המחלקה `Point`, עם שני שדות פרטיים ושדה סטטי, שסופר כמה נקודות ייצרנו (מופעים מהמחלקה). ניתן לראות למשל שהפונקציה `setX` מבצעת בדיקה של תנאי מעבר להשמה פשוטה לשדה, ועופפת את ההשמה הזו שמנצאת ב-X doSetX() שהוא מוגדרת private (הלקוק אמור להזכיר רק את X). המחלקה PointUser מייצרת שני אובייקטים מהטיפוס Point וקוראת לפונקציה `X.setX`. נשים לב:



של תנאי מעבר להשמה פשוטה לשדה, ועופפת את ההשמה הזו שמנצאת ב-X doSetX() שהוא מוגדרת private (הלקוק אמור להזכיר רק את X). המחלקה PointUser מייצרת שני אובייקטים מהטיפוס Point וקוראת לפונקציה `X.setX`. נשים לב:

-
-
-
-

במובן האובייקטיבים עצם נמצאים ב-Heap, ו-p1 מצביעים אליהם. במהלך ריצת הבניאי, הפניה this מצביעת על העצם שזה עתה הוקצתה. הפניה num הוא שדה סטטי ולכן גם הוא נמצא ב-Heap. בבניאי, היה שלב שבו **עוז**, אז מאותחלים ל-0, ואז התחבאה השמה לערכיהם שנשלחו בקריאה לבניאי, שהם 1 ו-2. בזמן של שירות מופע, עצם המטרה הוא העצם שעליו הופעל השירות. הפניה this תצביע לעצם זהה. בתוך הfonקציה .this.doSetX, הקריאה doSetX הוא בעצם setX, setX



סיכום ביניים:

- **שירותי מופע (instance methods)** – בשונה ממשירוטי מחלוקת (static methods) פועלם על עצם מסוים (this). בעודם משירוטי מחלוקת פועלם בדרך כלל על הארגומנטים שלהם.
- **משתני מופע (instance fields)** – בשונה ממשתני מחלוקת (static fields) הם שודות בתוך עצמים. הם מוצרים רק כאשר נוצר עצם חדש מחלוקת (ע"י new). בעודם משתנים גלובליים – קיימעו תוקן אחד שלהם, שנוצר בעת טיעינת קוד המחלוקת לדייבורו, ללא קשר ליצירת עצמים מאותה המחלוקת.

העברת פרמטרים, מחרוזות, מתחוזות (תרגול 3)

מחרוזות:

- באשר אנו מביצים `String new`, עברו שתי מחרוזות שונות עם אותו התוכן, נקבל שני מביצעים שונים למקומות שונים בזיכרון. לעומת זאת, אם לא נקצתה עצם חדש באמצעות `new`, אלא נכתב פשוט "hello" אז יכול להתבצע reuse של המיקום בזיכרון (באזור שנקרא **String Pool**).
- פונקציות שימושיות – `split`, `starts/endsWith`, `contains`, `indexOf`, `substring` וכמונן.



2 – מחלקות

מחלקות

חדים

עיצוב על פי חוזה (design by contract):

מצין בהערות התיעוד שימוש כל פונקציה:

- **תנאי קדם (pre-condition)** – מהן ההנחות של כותב הפונקציה לגבי הדרך התקינה שיש להשתמש בה.
- **תנאי אחר (post-condition)** – מה עשו הפונקציה, בכלל אחד מהשימושים התקינים שלה.

נשאטל לתאר את שני התנאים במנוחים של ביטויים בוליאניים חוקיים ככל שניתן (לא תמיד ניתן). שימוש זה הוא מדויק יותר. ויאפשר לנו בעתיד לאכוף את החוזה בעדרת כל חיצוני. התchapir שלנו מבוסס על כל בשם Jose.

```
/*
 * @pre denominator != 0 ,
 *        "Can't divide by zero"
 *
 * @post Math.abs($ret * denominator) <= Math.abs(numerator) ,
 *        "always truncates the fraction"
 *
 * @post (($ret * denominator) + (numerator % denominator)) ==
 *        numerator,
 *        "regular divide"
 */
public static int divide(int numerator, int denominator)
```

נשים לב:

- **מצב** - חוזה של שירות אינו כולל רק את הארגומנטים שלו. הוא יכול גם להתייחס לאיזשהו **מצב** של האובייקט (תמונה, זיכרון, ערכי משתנים) שරק בו ניתן לקרוא לפונקציה. למשל, במחלקה מסוימת קיימים שירות המattaח לבנייה נתונים, ושירות הקורא מאותו מבנה נתונים. תנאי הקדם של שירות הקוראה יכול להיות שמבנה הנתוניםobarot אוטומטית. נשים לב שימוש השירות getFib מתעלם לחולטי ממהקרים שבהם תנאי הקדם אינם מתקיים – **הימוש לא בודק את תנאי הקדם בגין המתוודה** (האם המחלוקת אוטומטית, האם העברתו צווקן).
- **שירות לעולם לא יבודק את תנאי הקדם שלו** – אם שירות בודק תנאי קדם ופועל לפי תוצאת הבדיקה, אז יש לו התנגדות מוגדרת היעב עבור אותו התנאי, כלומר הוא אינו תנאי קדם עוד. אי הבדיקה מאפשרת כתיבת מודולים "סובלניים" שיעטפו קריואות למודולים שאינם מניחים דבר על הקלט שלהם.
- **חלוקת אחריות** – החוזה מגדיר במנדיוק אחריות ואשמה, זכויות וחובות על כיצד המודול יעבד וכי צד ישתמש בו:
 - הלקוח – חייב למלא אחר תנאי הקדם לפני הקוראה לפונקציה.
 - בספק – מתחייב למילוי כל תנאי אחר (post) אם תנאי הקדם התקיים.

טענות על המצב:

בעת רצחה לבדוק את נכונות התוכנה. **משמעות (shemora, invariant)** הוא ביטוי בוליאני שערך נבן תמיד. נוכחים כי התוכנה ש滥נו בכונה ע"י כך שנגדיר עבורת משתנה, ומוכחים שערכו שנד בכל רגע נתון. להוכחה פורמלית (בעזרת לוגיקה) יש חשיבות מכך שאין מנתרלת את זו המשמעות של השפה הטבעית, וכן היא לא מניחה דבר על אופן השימוש בתוכנה.

ראינו דוגמה לשמורה, שטוונת טענה על משתנה גלובלי בשם counter. לא לקחנו בחשבון שניתן לשנות אותו גם מחוץ למחלוקת שבה הוגדר. בולמר, נכונות הטענה תליה באופן השימוש של הלוקחות בקוד. **לצורך שמירה על הנכונות יש צורך למנוע מלוקחות המחלוקת הגישה למשתנה counter**.

רראות פרטיטית (private):

הגדרת משתנה או שירות כ-private מאפשרת **גישה אליו ורק מ너ור המחלוקת** שבה הוגדר. שימוש ב-**private** "תוכם" את הבא, שיכול לקרות רק בתחום המחלוקת. בעת אם קיימת שגיאה בניהול המשתנה, היא לבטח נמצאת בתחום המחלוקת ואין צורך לחפש אותה בקרב הלוקחות. תיכון זה מכונה **הכמסה** (encapsulation). כדי ליצור גישה מובוקרת בכל זאת לשדה, נגדיר מתודה ציבורית **(public)** שתמחזיר את ערכו של המשתנה הפרטי – **היא מאפשרת חשיפה למידע, אבל לא כתיבה ושינוי של !counter**.



המשתמר הוא חלק מהחוצה של הספק כלפי הלקוח ולכן הוא מנוט בשפה שהלקוח מבין

```
/** @inv getCounter() == #calls for m() */
public class StaticMemberExample {

    private static int counter;

    public static int getCounter() {
        return counter;
    }

    public static void m() {
        counter++;
    }
}
```

- סוג נוסף - **משתמר הייצוג (representation/implementation invariant)** הוא משתמש המוביל מידע פרטיו.
- גם בתנאי בתר, עלולים להיות ביוטים פרטיים שנדרצה להסתיר מהלקוח (רלוונטי רק למפתחי המחלקה).

אמנם, אם מדובר במתודה ציבורית, אנו חייבים לציין את כל תנאי הקדם בר שהמשתמשים יכירו.

- ניתן למנוע גישה לשירות ע"י הגדרתו כ-private, הדבר מופיע פונקציית עדן פנימית של המחלקה, אין שום רצון לספק לחושף אותה בלבד הלקוחות.

מחלקות קיימות, מחרוזות, קבועים וקלט (תרגול 4)

מחלקה Scanner

سورק טקסט פשוט אשר יודע לחלק טיפוסים פרימיטיביים ומחרוזות. הוא "מחלק" את הקלט לרכיביו השונים.

- הוא יכול לאפשר לנו בפרט לקרוא קלט מהמשתמש. לשם כך علينا ליצור את האובייקט Scanner ולהעביר לו את ה-.System.in stream

```
public static void main(String[] args) {
    Scanner s = new Scanner(System.in);
    String[] fragments = s.nextLine().split(" ");
    String TranslatedText =
        Translate.execute(fragments[0], fragments[1], fragments[2]);
    System.out.println(TranslatedText);
    s.close();
}
```

- כאשר מבצעים nextLine לא מתחשבים ב-delimiter, מפסיקים רק את ח' הבא, ומדפיסים את כל מה שיש עד אלו. כאשר עושים next או nextInt מתחשבים ב-delimiter (אם לא הוגדר איז הדיפולטיו הוא רווח).

מחלקה File

- יש שימוש לב להבדל בין relative ל-absolute בהקשר לנתיבים. החיסרון ב-absolute הוא שנתיב במחשב הפרטி לא יהיה רלוונטי בסביבה אחרת שבה הקוד יירוץ. הקובץ ממוקם יחסית לתוכנית, ועדיף להשתמש ב-relative.
- כדי להתמודד עם ה-separator שמשתנה בין מערכות הפעלה, ניתן להיעזר ב-File.separator, או לקבל את המסלול בקלט מהמשתמש.
- כיוון שיכולה להיזכר שקיים מסווג FileNotFoundError, נצהיר על הטיפול בחיריג בחתימת הפונקציה שלנו. ככלומר נסיף throws Exception (החריג הכללי ביותר שיש).



```

private static final String FILE_NAME = "Software1/example5.txt";

public static void main(String[] args) throws Exception {
    Scanner s = new Scanner(new File(FILE_NAME));
    while (s.hasNextLine()) {
        String[] fragments = s.nextLine().split(" ");
        System.out.println(Translate.execute(fragments[0],
            fragments[1], fragments[2]));
    }
    s.close();
}

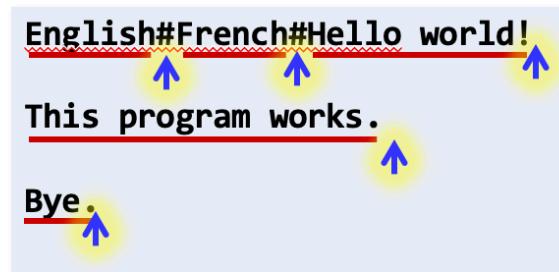
```

- נפרייד לפסקאות:

```

public static void main(String[] args) throws Exception {
    Scanner s = new Scanner(new File(FILE_NAME));
    s.useDelimiter("#");
    String srcLanguage = s.next();
    String destLanguage = s.next();
    s.skip("#");
    String text = "";
    while (s.hasNextLine()) {
        text += s.nextLine() + ' ';
    }
    System.out.println(Translate.execute(text, srcLanguage, destLanguage));
    s.close();
}

```



מדריך 10

פייצול מחרוזת לתחתיות-מחזרות:

המתודה split של המחלקה String מקבלת פרמטר שנקרא delimiter, שהוא סדרת תווים אשר מפרידה בין כל זוג תחת-מחזרות עוקבות. תחתיות המחרוזות אשר נמצאות במערך הפלט נקראו tokens. ה-parameter delimiter אינו מוכרכ להיות מחזרות פשוטה. השימוש הנפוץ ביותר הוא לפצל משפט למערך של מילים, כאשר המפריד הוא רווח לבן. ניתן לתפוס את כל הקומבינציות של רווחים, ירידות שורה וטאים על ידי הביטוי הרגולרי "+\s+". אם נרצה להימנע גם מהרווחים בקצוות המחרוזת, אפשר להשתמש במתודה trim().

זרמים (Streams):

זרמים הם קבוצה של טיפוסים שיודעים לקרוא מ- ולבתוב אל משבאים בצורה סדרתית. תמיד קוראים/כותבים בתים, הזרימה היא תמיד חד-כיוונית, ויש הפרדה בין זרמי Input ו-Output. כל הזרמים נפתחים עם יצירתם (FileOutputStream API) ווצר קובץ חדש. פתיחת זרם יכולה לגרום לריקת Exception. יש לסגור את הזרמים בגמר השימוש כדי לאפשר שחרור משבאים.

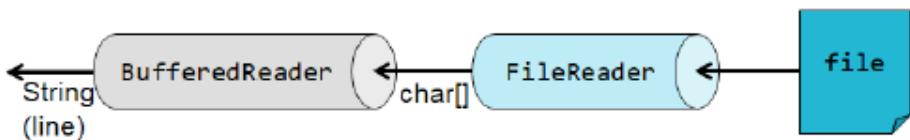
דוגמאות לשימוש בזרים:

שימוש	זרמים
קריאה/כתיבה של נתונים מקבצים	FileInputStream, FileOutputStream
קריאה/כתיבה של נתונים תוך שימוש במאגר מבנה	BufferedInputStream, BufferedOutputStream
קריאה/כתיבה של נתונים מקבצים	FileReader, FileWriter
קריאה/כתיבה של טיפוסים פרימיטיביים ומחרוזות (בדומה ל-Scanner)	DataInputStream, DataOutputStream



זרמים עוטפים – קיימים זרים אשר עוטפים זרים אחרים, ומוסיפים להם פונקציונליות. למשל, אם רציתם לקרוא מקובץ (BufferedReader) אבל שורה בכל פעם (FileReader) (BufferedReader).

`new BufferedReader(new FileReader(file))`



```

public class BufferedUnixToWindows {

    public static void main(String[] args) throws IOException {
        File fromFile = new File(args[0]);
        BufferedReader bufferedReader =
            new BufferedReader(new FileReader(fromFile));
        File toFile = new File(args[1]);
        BufferedWriter bufferedWriter =
            new BufferedWriter(new FileWriter(toFile));
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            bufferedWriter.write(line + "\r\n");
        }
        bufferedReader.close();
        bufferedWriter.close();
    }
}
  
```

אנו נתמקד בדרמים העוטפים - BufferedReader, BufferedWriter, היות והם השימושים ביותר לצוריכינו בקורס. אלה הם זרים עם מאגר מובנה (buffer) שמנוהל אוטומטית, BufferedReader יכול לקרוא תווים ממאגר המובנה, אך שבקריאות הבאות, ולחסן את התווים הנוספים במאגר המובנה, במקום לגשת שוב לדיסק הוא יוכל לשולף אותן מהמאגר המובנה. זה עדיף כיון שהוא של גישה לדיסק הן היקרות ביותר, ועוד יותר קראוא/לבוט בכמה שפותות פעמיים.

בעת נתבונן שוב בתכנית שמירה קבצים מיינקס לוינדוס, הפעם עם זרים עוטפים. נשים לב למספר הבדלים חשובים. ישנו יתרונותבוליטים בנוחות השימוש, מעבר לשיפור ביצועים.

ראשית, בעת נתן לעובוד ישירות עם מחרוזות (בל' מעבר מסורבל ממערך `char`), ולקראא שורה שלמה (עד שנקלים בתו `\n` או `\r`) בפקודה אחת – `readLine()`. מתחודה זו מחזירה מחרוזת המכילה את השורה שנקרה, ואם הגיענו לסוף הקובץ מוחדר הערך `null`.

הדוגמה הזאת משמשת בתבנית בסיסית עבור קלט/פלט עם קבצים באמצעות זרים.

השוואה בין Scanner לזרמים:

ל-Scanner יש Buffer, אבל הוא קטן יותר מה-Buffer של BufferedReader. גודל ה-Buffer רלוונטי באשר לעבוד עם קבצים גדולים ונרצה לחסוך גישות לדיסק. לעומת מעט קריאות של הרבה בתים, מאשר הרבה קריאות של מילים בלבד. Scanner מאפשר פועלות עיבוד מתחכמת על קובץ הטקסט אותו אנו קוראים, ומפרק את הקלט ל-tokens. בעוד ש-BufferedReader מחזיר מחרוזות בלבד, ה-Scanner יכול להחלץ טיפוסים פרימיטיביים כמו int, Boolean וכו'. זה שימושי כאשר אנחנו רציתם לבצע המרות תוך כדי הקריאה מהקובץ.

המחלקה StringBuilder:

```

StringBuilder sb = new StringBuilder("abc");
sb.append("d");
System.out.println(sb);
  
```

באשר אנו רציתם לבצע ברכף שינויים רבים על אותה המחרוזת, ניעזר במחלקה זו כדי למונע בעיות ייעילות זמן וזיכרון. אם אנחנו מבצעים בולולה הרבה שינויים, כל הזמן נוצרים מופעים חדשים של מחרוזות הבניים ויש צורך להעתיק כל פעם מחרוזות חדש. במקרים אלו,

ניתן לבעוקה או שיריכולה ליציג מחרוזת שהיא mutable – ניתן לבצע בה שינויים מלבlich ליצור אובייקט חדש בכל פעם. במקרה אופרטורו השרשורי משתמש במתודה append. ניתן להמיר בכל שלב למחרוזת על ידי המתודה `toString()`.

הנה קוד בסיסי אשר מוסיף למחרוזת abc את הtau p ומדפיס את התוצאה:

מחלקות, עצמים, חודם (תרגול 5)

בקורס הנוכחי אנחנו מאפשרים גמישות בתחריר של כתיבת חודם

מופע מחלקה:

- **ניתן להשתמש ב:**
 - **תנאים בוליאניים בגאווה ($0 \leq x$)**
 - **תגיוט מהסוגן (שנלמד בהרצאה):** ("M is a diagonal square matrix")
 $(x \in [0,1])$
 - **ביטויים ונוסחאות מתמטיים (x)**
 - **שפה חופשית ("M is a diagonal square matrix")**
 - **שלובים של הנ"ל, ועוד**
- **בכתיבת חודם חשוב לשמר על התיקנות לכל המקרים שמתאים לתנאי הקדם בתנאי الآخر**
- **תמציתי, בהיר ומדויק!** (ביחוד אם משתמשים בשפה טבעיות)
- **טיפול בקלט שלא עומד בתנאי קדם הוא מיותר ולא רצוי, אך לא נחשב רשותית להפרת חוזה!**

- שודות מופע יהיו לרבות עם הרשות גישה private.
- כאשר נכתב חוזה ב מבחן, בתנאי הקדם הוא לא יודע שיש שדה בשם balance, getBalance-ב-balance.
- בנייתו לא אמור לכלול לוגיקה פרט לתחול שודות המופע. לאחר האתחול העצם חייב לקיים את משתמר המחלקה.
- ניצבר בתחריר () של קרייה לבניין אחר של אותה מחלקה, ניתן להשתמש בה רק מtower בניין.

Instance vs. Class (static) Fields

Instance fields	Class (static) fields
למה? <ul style="list-style-type: none"> ■ ייצוג פנימי של המופע 	למה? <ul style="list-style-type: none"> ■ קבועים ■ ערכים המשותפים לכל מופע המחלקה
מתי? <ul style="list-style-type: none"> ■ מואתחלים עם ייצרת האובייקט 	מתי? <ul style="list-style-type: none"> ■ מואתחלים לפי הסדר עם טיענת המחלקה
כמה? <ul style="list-style-type: none"> ■ אחד לכל מופע 	כמה? <ul style="list-style-type: none"> ■ יש רק 1 בכל התוכנית! (0 לפני טיענת המחלקה)
מאיפה? <ul style="list-style-type: none"> ■ נגישים אך ורק ממетодות מופע (למה?) 	מאיפה? <ul style="list-style-type: none"> ■ נגישים ממетодות סטטיות ומethods מופע

28



מנשכים

מנשכים (Interfaces)

הגדרת מנשכים:

מה המנשך יכול להכיל?

```

public interface MyInterface{
    int i = 0;
    public static final int j = 0;

    void func1();
    public abstract void func2();

    default void defaultFunc(){
        func1();
        privateFunc();
    }
    private void privateFunc(){
        System.out.println("a!");
    }

    static void staticFunc(){
        System.out.println(i);
    }

    private static void printI(){
        System.out.println(i);
    }
}

```

1. **משתנים –** שדות גלובליים קבועים, **public methods** (פונקציות בונראות **public**)
2. **שירותי מופע ציבוריים (public methods) –** פונקציות בונראות **public** **(abstract)** – לא מימוש (default) – שירות מופע (פונקציה שהיא לא סטיטית) שהוא **default** הוא **שירות מופע** הממושב בתוך המנשך (בניגוד לשאר הפונקציות שהוא פונקציות מופע לא מימוש), ובמידה מסוימת מדובר בסתירה לקונספט המוקורי של מנשך. עם זאת, הכוון של פונקציות באלו מוגבל, כיון שאין להן גישה לשדות מופע, הקיימים במימושים הקונקרטיים של המנשך.
3. **שירותי מופע (עם מימוש) –** בונראות **private**. השירות זה יכול לשמש כשירות עזר לפונקציות **default** שהן תמיד **public**.
4. **שירותים סטיטיים (עם מימוש) –** בונראות **public/private**. בפונקציות הללו ניתן לגשת לשדות סטיטיים שמוגדרים במחלקה. בדיפולט הנראות היא **public**.

מה המנשך לא יוכל?

1. שדות מופע.
2. בנאים.

ראינו את המנשך **Point**, שמצוור על פונקציות רבות. המנשך הוא המקום האודיאלי להגדרת חזה ומצב מופשט, מכיוון שהוא נטוניים טרם נכתבו, זה המקום שבו ניתן כתוב דברים כלליים שאינם קשורים למימוש ספציפי זהה או אחר.

סוגי פונקציות:

- **שאילות צופות (observers) –** פונקציות שמסתכלות על האובייקט ומחזירות חוווי כלשהו לגבי המצב שלו. הערך המוחזר אינו מהטיפוס שעליו הן פועלות. פונקציות אלו לא משנות שום דבר באובייקט.
- **שאילות מפיקות (producers) –** פונקציות אלו מחזירות אובייקט חדש. הן לא משנות את העצם הנוכחי, ומחזירות עצם מהטיפוס שעליו הן פועלות.
- **פקודות –** ביצוע פעולה על ידי האובייקט שמייצרת شيئا' כלשהו.

כל אחת מהפונקציות שאנו כותבים, כדי שתשתתף לאחת מהקטגוריות (צופות, מפיקות, פקודות). כאשר פונקציה שייכת ליותר מקטגוריה אחת (למשל סוק במחסנית)>Katz יותר מורכב להבין אותה. נשתדל שמצב בה לא יקרה (אבל זה קורה לעיתים).

נקודות בעיינות במימוש

לאחר שהגדכנו את המנשך Point, ניתן למשתמש בשרט Rectangle בלבד, ללא תלות במימוש קונקרטי כלשהו שלו. ואכן, נוכל למשתמש נקודה בצורות שונות על בסיס אותו המנשך. בתיבה נdanaה של שירות המלבן תעשה שימוש בשירותי הנקודה.

נקודות בעיינות במימוש:

- **מימוש חלקי –** התחלנו למשתמש בשרט Rectangle באמצעות הפונקציות שבמנשך Point ונתקלנו במספר פונקציות שאנו לא יודעים בעת כיצד למשתמש, בעיקר כי צריך להחדיר נקודה חדשה, ול-Point אין בכך! לכן לא נוכל לייצר נקודה חדשה בהתאם למטרת הפונקציה.
- **שדות פנימיים חשופים –** זההינו מקומות בעלייתים, במבנה ובחוזה של הנקודות. כמו שמדובר בהפניות, אנו מבצעים השמה למביע (ומצביעים לאותה הכתובות שספקה לנו) או מחזירים מביע לשדה שאמור להיות פנימי במחלקה. אך נוכל להשפיע על נקודות במימוש הפנימי של המלבן, מהוז למחלה!
- **משתמר המלבן –** אם היינו מנסחים בזירות את משתמר המלבן, היינו מגלים כי עברו מלבן שצלעותיו מקבילות לציריהם, צריך להתקיים בכל נקודת זמן:



```
/** @inv bottomLeft().x() < bottomRight().x()
   @inv bottomLeft().y() < topLeft().y()
*/
```

מציג כמה דרכי ה证实 (Validation) עם הבעיה:

1. **התעלמות** – אם אנו משוכנעים כי לא יעשה שימוש לרעה בערך המוחזר, ניתן להשאיר את המימוש כך. הדבר מסוכן ולא מומלץ, אולם אם השימוש בחלוקת מוגבל (לדוגמה – רק על ידי מחלוקת מסוימת), ניתן לוודא כי כל השימושים מכבדים את משתמר המלבן.
2. **נקודה מקובעת (immutable)** – הגדרת **חלוקת שאין לה פקודות כליל**. במקרים שהבעיה התגלתה במלבן, אנו פותרים את הבעיה ע"י שימוש הנקודה, **את הפקודות יחליפו מפיקות אשר צרו עבור כל שינוי מבוקש עצם חדש**, עם התוכונה המבוקשת.חלוקת היא מחלוקת כזו – למשל `toUpperCase` מחליף הפינה לעצם חדש, **לא משנה את המחוות עצמה**. הבעיה בכך – זה משנה את הייעוד שלחלוקת, לא יכול לעשותו אותה שום דבר.
3. **שיבוט הנקודה (clone)** – נוסיף `IPoint` מפיקה משכנת, הכולרת נסף שירות בשם `clone` אשר ייחזר העתק של העצם הנוכחי. בך לא יוווצר מצב שמשהו מבוצע וכל מצביע לשדה הפנימי. הבנאי אשר מקבל נקודות בארגומנטים ישים את השיבוט שלו, וכן גם המתוודות `topRight`-`bottomLeft` ו-`topLeft`-`bottomRight`. **לכן שינויים על הערך המוחזר לא ישפיעו על הקודקוד המקורי.**

כללים מנחים למימושחלוקת מקובעת (immutable):

```
public record Point(int x, int y){  
    //methods here  
}
```

השדות מוגדרים כ `final`. אין שום דרך לעדכן אותם לאחר היצירה, ולכן האובייקט הוא `immutable`. מה היה קורה אם אחד השדות היה מטיפוס `[int]`?

- כל השדות יוגדרו `private final`.
- אין שום פונקציות שמעודכנות את ערכי השדות.
- אם השדות הם טיפוסי הפינה שאינם מקובעים (לא מחזוחות) – אין מתודות שמחזירות מצביעים לשדות אלה. אם צריך להחזיר את השדה, יש להחזיר עתק.
- לא לשמרו מצביע לאובייקט לא מקובע חיצוני. אם צריך, לייצר עותקים ולשמור אותם.
- אין לאפשר ירושה.

שימוש באמצעות record מתחאים-immutable. אבל לא כל `record` הוא בהכרח `immutable`. אם אחד השדות בחלוקת היה מערך `[int]`, הפונקציה שמחזירה את א' חושפת אותו למשתמש וניתן מבוצע לבצע בו שימושים, זה טיפוס הפניה.

הערות נוספות על טיפוסים שהם mutable:

- כאשר יותר מכך מחייבים ל Koho מושגים משתמשים בעצם מטיפוס `mutable`, יש לברר האם שינוי העצם ע"י אחד הלkopoot אמור להשפיע על כל הלkopoot. אם לא, יש לספק העתק.
- גישה אחרת סובرت כי יש להגדיר בעלות (ownership) על עצמים מסווג זה. בך ידע כל לקוח מה מותר לו לעשות ומה אסור לו, ובמקרה הצורך ייצור עצמו העתק.
- עבודה עם טיפוסים שהם `immutable` גורמת יצירתיות עצם חדש עבור כל שינוי. כאשר הדבר מתרחש בצורה תכופה (למשל בתוך לולאה שרצה הרבה), התוכנית מייצרת הרבה זבל.
- עבודה עם טיפוסים שהם `mutable` מאפשרת מחזור של עצמים ע"י Object Pooling.

רב-צורתיות (Polymorphism)

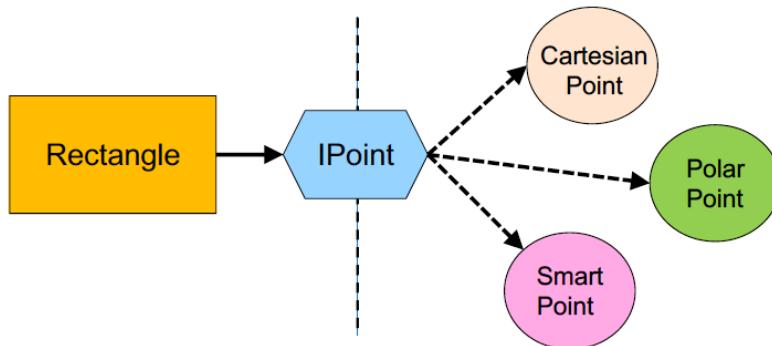
נראה 3 מימושים אפשריים למנשך `IPoint`. כל מימוש כזה נכתב באמצעות `IPoint XXXX`. **כל מימוש קונקרטי בזה הוא סוג של `IPoint`** (כי הוא מממש את כל הפעולות של `IPoint`). ניתן להוסיף בכלחלוקת בזווית שמאםת את המنشך, פונקציות נוספות שלאין חלק מהמנשך.

1. **נקודה קרטזית (Cartesian):** נקודת המיצגת על ידי שתי קואורדינטות – y , x . קיימן `tradeoff` בין מקום וזמן. תוכנה שנשמרת בשדה (y , x) תופסת מקום בזיכרון אך חוסכת זמן גישה, ולעומת זאת תוכנה שמאםת כפונקציה (`rho`, `theta`) חוסכת מקום אך דורשת זמן חישוב בכל גישה.
2. **נקודה פולארית (Polar):** במרקחה זה אנו שומרים את `rho`, `theta` בתור שודות, ואת y , x מקבל דרך פונקציות.
3. **נקודה חכמה (Smart):** בmäßigוש זה אנו שומרים את כל השודות, ובוליאנים של `cartesian`, `polar`, `rho`. בוליאני שערכו `True` מסמן שהציגו בסוג זהה של הנקודה הוא תקין. המשתמר הוא `tau_kmp` כיוון שמדובר במימוש פנימי.



```
/** @imp_inv polar | | cartesian , "at least one of the representations is valid"
 *
 *  @imp_inv polar && cartesian $implies
 *          x == r * Math.cos(theta) && y == r * Math.sin(theta)
 */
```

מחלקה זו לא מבצעת חישובים שהוא לא צריך לעשות – נחשב שיעורים רק בסוג מסוים של נקודה, ולא נחשב בסוג השני. נודא שישיעורים אלו יסומנו ללא עקביהם בסוג השני (לכן נעדכן את הסוג השני ל-false), ובמקרה הצורך נעדכן אותם בעתיד.



אבחנה – כל `CartesianPoint` הוא גם `IPoint`! אבל לא כל `IPoint` הוא גם `CartesianPoint` (הוא יכול להיות למשל `PolarPoint`). בעת נשים לב לשימוש במנשכים. (`PolarPoint`

```
IPoint polar = new PolarPoint(Math.sqrt(2.0), (1.0/6.0)*Math.PI);
```

הטיפוס של המשתנה `polar` – הוא `IPoint`. הטיפוס של האובייקט עליו מציין `point` – הוא `PolarPoint`. כמובן, אנחנו יכולים לראות שמשתנה מטיפוס `X` יכול להצביע על אובייקט מטיפוס `Y` אשר שונה מ-`X`. זה לא אפשרי לכל `X` ו-`Y`, אלא רק לכאליה שמתקיים ביניהם יחס מיוחד. למשל אם `Y` (`PolarPoint`) הוא סוג של `X` (`IPoint`): מתקיים כי `X` מטיפוס מנשך, `Y` ממשם את המنشך.

```
System.out.println(polar.toDegreeString()); // Compilation Error
```

בזמן קומpileציה אנו לא יודעים לאיזה אובייקט `polar` מצביע, לכן אי אפשר לקרוא לפונקציה `toDegreeString()`. בולמרן – בולמר אמר יודעים רק **שהטיפוס של המשתנה הוא מסווג `IPoint`,** לכן נוכל לקרוא **לפונקציות מהמנשך של `IPoint` בלבד.** התיקון – נקרא לפונקציה `toString`, ששייכת ל-`IPoint`.

```
IPoint point = new IPoint (1.0, 1.0); // Compilation Error
```

גם קריאה לבנייא אינה אפשרית, כי אין לנו מנשך בנאי בפני עצמו.

לסיכום:

- ניתן להגדיר ב-Java הפניות מטיפוס מנשך.
- הפניות אלו יקבלו בפועל השמות לעצמים מחלקות המMESSות את המنشך. השימוש ההפוכה אסור! לא ניתן לבצע השמה של הפניה מטיפוס מנשך, להפניה מטיפוס מחלקה קונקרטית שMESSת אותן:

`CartesianPoint cartesian = ...`

`IPoint point = ...`

- `cartesian = point;`
- `point = cartesian;`

השורה הראתה לא מתקמלת, כי מדובר בעצמים של מחלקות שונות עם פונקציות שונות. **אפשר שהכללי מצביע לפרטי (הمرة כלפי מעלה), אבל לא שמקורה פרטי מצביע למקורה כלל!** (הمرة כלפי מעלה – רק במקרים מסוימים).

על עצמים אלה ניתן יהיה להפעיל בעדרת המنشך רק שירותים שהוגדרו במנשך עצמו.

למנשכים אסור להגדיר בניאי, ולא ניתן ליצור מהם עצמים.

בכתיבת תוכנה נשתדל להגדיר משתנים מטיפוס המنشך, כדי לצמצם ככל הנימן את התלות בין הקוד המשמש והמיומש של אותן מחלקות.



פולימורפיزم (polymorphism) – באשר במלבן נמשמעות translate, נפעיל את translate על הנקודות, מוביל לדעת איזו נקודת זו באמצעות (מהו המימוש הקונקרטי, מאי זה סוג של נקודת הוא). המלבן שלו יודע לעבוד עם כל מחלקה שemmמת את הממשק IPoint!. הפונקציונליות שתקרה בפועל, היא של הנקודה הספציפית, העצם הספציפי שהמלבן יכול.

```
public interface I1 {
    public void methodFromI1();
}

public interface I2 {
    public void methodFromI2();
}

public interface I3 {
    public void methodFromI3();
}
```

```
public class C implements I1, I2, I3 {
    public void methodFromI1() {...}
    public void methodFromI2() {...}
    public void methodFromI3() {...}
    public void anotherMethod() {...}
}
```

ריבוי מנשכים – מחלוקת אחת יכולה לממש במא
מנשכים. במקרה זהה כל אחד מהמנשכים מבטא הובט/תוכנה של המחלוקת. למנשכים אלה בדרך כלל מספר מצומצם של מתחזות. השמה של מחלוקת קונקרטית לתוך הפניה מטיפוס מנשך שכזה, מהו זה הטלה של המחלוקת על מישור התוכנה שאotta מבטא המענק (narrowing).

אם למשל במנשך I2 שם הפונקציה גם היה methodFromI1 (וערך החזרה ע"י void), לא הייתה בעיה למשוך את כל המנשכים, כי I1 ו-I2 מגדירים אותה הדירשה, שאotta נמלא במימוש של C (מדובר באותו חותימה). מדובר בראשית חבות שעילנו לממשק, **אם אף שני מנשכים ביקשו את אותה חותימה – עלינו לממש אותה בדיקוק פעם אחת זה תקין!**

מבנה עיצוב המפעל (Factory Design Pattern)

מבנה עיצוב היא פתרון מקובל לבניית תוכן נפוצה בתוכנות מונחה עצמים. היא מתארת כיצד לבנות מחלוקות כדי לענות על הדירשה הנתונה, ומספקת מבנה כללי שיש להשתמש בו כשמממשים חלק מתוכנית. היא לא מתארת את המבנה של כל המערכת, או אלגוריתמים ספציפיים. היא מתמקדת בקשר בין מחלוקות.

נזכיר בבעיה שהיא לנו בבניין של המלבב, ובמתחודה bottomRight:

- **ניסיוון 1** – נגיד במנשך IPoint createPoint () אשר ימומש בכל אחת מהמחלקות הקונקרטיות, כאשר הוא ייצור נקודת חדשה ויחזר אותה. אמנם, כדי להשתמש במתודה יש להפעיל אותה על עצמים שנוצרו בבר, ובבנייה של המלבן עד לא נוצרה אף נקודת.
- **ניסיוון 2** – נגיד את המתודה בסטטיות static IPoint createPoint () . אכן, המלבן צריך ליצור נקודות, אבל הוא מכיר רק את IPoint, ולא את המימושים הקונקרטיים.

```
public class PointFactory {

    public PointFactory(boolean usingCartesian, boolean usingPolar) {
        this.usingCartesian = usingCartesian;
        this.usingPolar = usingPolar;
    }

    public PointFactory() {
        this(false, false);
    }

    public IPoint createPoint(double x, double y) {
        if (usingCartesian && !usingPolar)
            return new CartesianPoint(x, y);

        if (usingPolar && !usingCartesian)
            return new PolarPoint(Math.sqrt(x*x + y*y), Math.atan2(y, x));

        return new SmartPoint(x, y);
    }

    private boolean usingCartesian;
    private boolean usingPolar;
}
```

נסתכל על רכיב חדש שהוא PointFactory. ביוון ש-Point לא יודע לייצר נקודות, ה-Factory יחולט איזו נקודת להחזיר. נגיד מחלוקת בשם PointFactory שתבל מתחודה שתפרקודה יהיה כמו תהיה שדה במחלוקת של המלבן.

יש לנו כאן בנאי ברירת מחדל שקורא לבניין הראשון עם false, false – מה הערכים שקובע את המצב להיות יצירהSmartPointer. הפונקציה של createPoint SMARTPOINT היא זו שתחזיר נקודת לפי הסוג הרלוונטי. הסוג נקבע בהתאם לשדות בוליאניים usingXXX שקיימים במחלוקת זו.





מדוע שימוש במפעלים עדיף? הרי בעצם יש תלות בין המפעל ובין מחלקות הנקודות הקונקרטיות:

- מחלוקת המלבן היא **מחלקה כללית**, המיועדת לשימוש נרחב עם מגוון נקודות שverb נכתבו ושורם בכתביו.
- מחלוקת המלבןוסף על היותה ל Koh של המנשך Point, משמשת גם **ספק כלפי צד שלישי** (שריצה ליצור מלבים).
- **נקודות המלבן**, הם אלו שצרכים להזכיר את מגוון הנקודות הזמן לשימוש. מחלוקת המפעל חוסכת מהם את ההטעסקות **בפרטים אלה**.
- שימוש במפעלים מדגיש את ההבדל בין **הידע שיש לנוטב הספרייה**, לעומת **הידע שיש לנוטב האפליקציה**. זמינות המימושים תהיה ידועה במלואה רק בזמן קונפיגורציה.

מנשכים, פולימורפים (תרגול 6)

- יכולם להיות שירותו מופיע בנתאות **private – חדש!** יכול להופיע ב מבחנים קודמים בתור שהוא שאינו אפשר.
- ראיינו את המנשך Shape ושתי מחלוקות שemmashו אותו – טיפוס הפניה מסוג המנשך Shape יכול להציגו אל כל אובייקט המழמץ את המנשך. ניתן לקרוא באמצעותו רק **למתודות הכלולות בהגדרת המנשך**. כדי לקרוא למתודה **הספציפית** למחלוקת (Circle), בבעוד **downcast**
- **upcast – cast** – הוא אפשר לעשות במרומות, downcast חיבים לעשות בבירור. לרבות נימנע מהמרות באופן כללי. **יכולת לקרות שגיאת זמן ריצה, אם נציג downcast לטיפוס שאינו תואם את הטיפוס בפועל** (זה יתקומפל, אבל בזמן ריצה תהיה שגיאה בשיתוברה שהטיפוס אינו תואם).
- ראיינו דוגמה נוספת של נג3mp, לאחר מכן נרצה שינגן גם video. במקום לשכפל קוד, וכותבו מנשך Playable ופונקציה play. שתי מחלוקות שונות יממשו את המנשך. יוכל לכתב פונקציה שבולולה תנגן item בצדורה פולימורפית (זמן ריצה הטיפוס הדינמי יבריע אותה מימוש של play ויקרא).

```
public void play (Playable[] items) {
    do {
        if (shuffle)
            Collections.shuffle(Arrays.asList(items));

        for (Playable item : items)
            item.play();

    } while (repeat);
}
```

```
public interface Playable {
    public void play();
}
```

- מחלוקת יכולה למש יותר ממנשך אחד.
- **מנשך יכול להרחיב מנשך אחר** (ואז יכול גם את המתודות המוגדרות במנשך זה). **הוא יכול גם להרחיב יותר ממנשך אחד**, גם אם מופיעה בשני מנשכים או יותר הצהרה על אותה פונקציה אבסטרקטית.
- כדי להמיר מספר ליצוג הבינארי שלו ניעזר בפונקציה **toBinaryString**.

- לא ניתן ליצור מופיע של מנשך בעזרת הפקודה new.
- מנשך יכול להכיל מתודות וגם קבועים
- מחלוקת יכולה למש יותר ממנשך אחד באווה (תחליף לירשה מרובה).

```
public class Circle implements Shape, Drawable {...}
```

- **מנשך יכול להרחיב מנשך אחר** (ואז יכול גם את המתודות המוגדרות במנשך זה).

```
public interface Shape extends Drawable {...}
```

טיפוסים גנריים ומחלקות פנימיות

מבנה נתונים מקוثر

כדי לייצג מבנים מקוזרים (כגון רשימה מקוורת, עץ, וכו'), מגדירים מחלקות שכוללות שדות שמתיחסים לעצמים נוספים מאותה מחלקה. כדוגמה פשוטה, נגדיר מחלקה `IntCell` שעצמאית בה מייצגים איבר ברשימת מקושות של שלמים.

- המחלקה מייצאת בנאי לייצור עצם, כאשר התוכן והאיבר הבא הם פרמטרים.
- המחלקה מייצאת שאילתות עבור התוכן ועבור האיבר הבא, ופקודות לשינוי האיבר הבא, ולהדפסת תוכן הרשימה מהאיבר הנוכחי.
- אפשר היה למשוך את המחלקה זו בטור `record` אבל מה שהורס פה זה הפונקציה `setNext`. כי ב-`record` כל השדות הם `final`.

```
public class IntCell {

    private int cont;
    private IntCell next;

    public IntCell(int cont, IntCell next) {
        this.cont = cont;
        this.next = next;
    }

    public int cont() {
        return cont;
    }
}
```

```

graph LR
    A[cont: 1  
next] --> B[cont: 2  
next]
    B --> C[cont: 3  
next]
    C --> null
    classDef IntCell
  
```

הכללת טיפוסים (Generics)

הטיפוס הגנרי:

הרשימה הנוכחית עבדת עם טיפוס `int` בלבד, ונרצה להכליל את הטיפוס **שאינו** היא עובדת, **לטיפוס גנרי**. זאת כדי להימנע משכפול קוד ביצירת מבנה מקוושר מטיפוס אחר – תווים, מחרוזות, וכו'. מחלקה גנרית מגדירה טיפוס גנרי, שמצוין אחד או יותר משתני טיפוס (type variables) בתוך סוגרים משולשים. נכליל את המחלקה `IntCell<T>`, כלומר תוכן התא יהיה מטיפוס פרמטר `T`, אך ככל התאים ברשימה הם מאותו הטיפוס `T`.

מה השתנה במחלקה?

```
public class Cell <T> {

    private T cont;
    private Cell <T> next;

    public Cell (T cont, Cell <T> next) {
        this.cont = cont;
        this.next = next;
    }
}
```

- בכותרת המחלקה נוסף משתנה הטיפוס `T`. הערה: `T` זו בחירה שירוטית, המשתנה יכול להיות בעל כל שם!
- הטיפוס שמוגדר הוא `<T>`.
- הטיפוס של כל שדה/משתנה זמני/טיפוס מוחזר של השירות שהוא `int` יוחף ב-`-T`.
- הטיפוס של כל שדה/משתנה זמני/טיפוס מוחזר של השירות שהוא `Cell` יוחלף ב-`<T>`.
- לשום לב להבדל – בכותרת המחלקה מגדירים פרמטר בשם `T`, ובשדה משתמשים בפרמטר שהגדכנו `T` (לא יכולים להשתמש בפרמטר אחר, רק `T` הוגדר).

כדי להשתמש בטיפוס גנרי, יש לספק בהצהרה על משתנה **בקריאה לבנייה** (במישר הקורס נראה שאפשר להימנע מכך), טיפוס קונקרטי עבור כל משתנה טיפוס שלו, למשל `Cell <String>`.

טיפוסים עוטפים (Wrappers)

הטיפוס הקונקרטי שMOVED במקומ הטיפוס הגנרי, חייב להיות טיפוס הפניה, כלומר אינו יכול להיות פרימיטיבי. למשל, לא יוכל לכתוב `<int>.Cell`. לצורך זה נדרש לטיפוסים עוטפים.

לכל טיפוס פרימיטיבי קיים טיפוס הפניה מתאים: float-Float, double-Double, int-Integer, char-Character. כל הטיפוסים העוטפים הם `immutable`. הם שימושיים כאשר יש צורך בעצם (בטיפוס הפניה).

- `Character` – תרגום טיפוס פרימיטיבי לטיפוס העוטף שלו, ע"י קרייה לבניית המתאים. למשל מ-`char` ל-`Character`.
 - `Unboxing` – תרגום טיפוס עוטף לטיפוס הפרימיטיבי המתאים.
- Java מאפשרת מעבר אוטומטי בין טיפוס פרימיטיבי לטיפוס העוטף שלו:

```
Integer i = 0; // autoboxing
int n = i;      // autounboxing
if(n==i)        // true
    i++;         // i==1
System.out.println(i+n); // 1
```

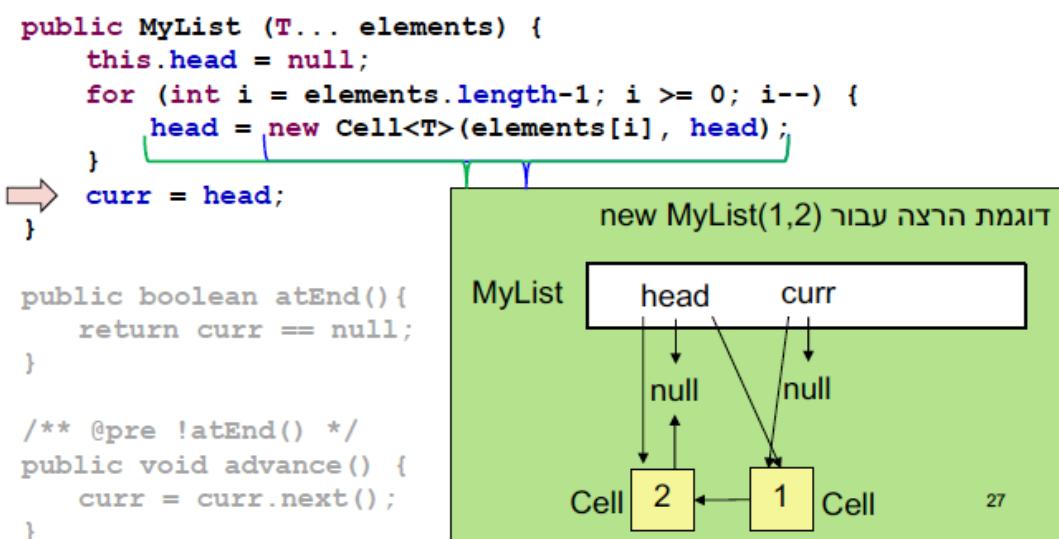
אם כך, מדוע אנו לא משתמשים רק בטיפוסים עוטפים בשפה, ואז הינו מתעסקים אך ורק בהפניות? כי זה לא חסכו בזיכרון, וכן פחותה חסכוני בזמן ריצה – צריך למכת מקום בזיכרון ולמצוא שם בתובות, שבמוקם שלא בזיכרון מתגלה התוכן הרלוונטי. בטיפוס פרימיטיבי, הכתובות שלו מכילה את המידע עצמו.

עיצוב הראשינה:

בעת נüberו לעסוק ב-`MyList<T>`. הוא מייצג באמצעות רשימה מקוּשָׁת, והוא מייצג רק תא בודד (כמו `Node` שראינו מבוא). ניתן וצריך לבטא את שני הרעיוןֹת – רשימה, ותא – בטיפוסים בשפה עם תכונות המתאימות לרמת ההפשטה שלהן. נציג את המחלקה `MyList<T>` המייצגת רשימה:

1. ביסיון ראשון – נגדיר שדה פרטי מטיפוס `T Cell`, **שি�בע** לראש הרשימה. הבעה – מימוש הרשימה אמור להיות חלק מהייצוג הפנימי שליה ומושתר מהלוקה, ואילו במימוש זה הלקוחות צריכים להזכיר גם את `Cell`. הדבר פוגע בהפשטה המחלקה של רשימה מקוּשָׁת.
2. ביסיון שני – נגדיר בנוסף שדה פרטי מטיפוס `T Cell`, **שি�בע** לאייר הנוכחי ברשימה. בעת הלקוח לא צריך להזכיר את `Cell`, השימוש הוא פנימי למימוש בלבד.

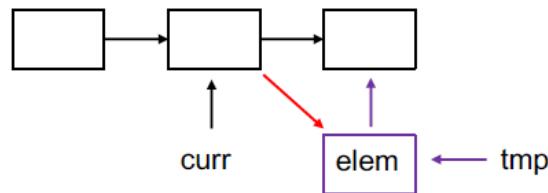
בבנייה `MyList` אנו מקבלים מספר בלתי מוגבל של איברים מטיפוס `T`. **הולאה הולכת הפוך**, עוברת על רשימת האיברים מהסוף להתחלה. זאת ביוון שההפשטה האיברים לרשימה, מתבצעת על ידי קרייה לבניית `Cell` עם האיבר הנוכחי שעוברים עליו, כאשר האיבר הבא שהטה החדש **צבע** אליו, הוא האיבר הנוכחי שהוא בידינו. ככלומר נניח שהווסףנו את 5, ואז את 4, אז 4 יצביע ל-5. لكن הולאה בסופה של דבר תיצור רשימה של האיברים המבוקשים מהתחלה לסוף, והם יצביעו אחד לשני כמו שציריך.





הfonקציית addNext מוסיפה חוליה חדשה לרשימה:

```
/** @pre !atEnd() */
public void addNext(T elem) {
    Cell<T> temp = new Cell<T>(elem, curr.next());
    curr.setNext(temp);
}
```



איך נוסיף את הפונקציה (x) addHere(int x) – הוספת האיבר x **למקום הנוכחי ברשימה?**

סוגיה זאת קצת יותר מורכבת, כי אין לנו מצביע לאיבר שנמצא אחד לפני המקום הנוכחי. גישות אפשריות:

1. תחזקה של prev נוספת על curr.
2. נróż מתחילה הרשימה עד המקום אחד לפני הנוכחי.
3. החלפת תכני התאים.

גישה 1 ו-2 פשוטות רעיוןית אך פחות אלגנטיות (תחזקה, ביצועים). ננסה למשת את גישה 3, בולם נבצע addNext לתא עם הערך החדש, ואז נחליף בין ערכי התאים (אין שום דרישת-cell מסוים יהיה במקום שבו הוא נמצא, אכפת לנו רק מהערבים...).

בעיה – **איך אפשר לעדכן את השדה cont של ה-cell כי הוא private!** עם זאת, אם רצים לאפשר גישה של myList לשדה הפרט של Cell, המחלקה Cell היא מחלוקת עוז של myList וכן יש הצדקה למתן הרשות גישה חריגות באלו.

- לא ניתן להוסף מתודה setCont() למחלקה Cell, יכול להיות שזה יוצר באגים במקומות אחרים.
- אם שתי המחלקות היו באותו package, אפשר להשתמש בнерאות חבילת – אבל אז כל מחלוקת אחרת בחבילה תוכל גם היא לגשת לפריטים אלה של Cell.

ניתן להגדיר אינטימיות בין מחלוקות ע"י הגדרת אחת המחלוקות במחלוקת פנימית של המחלוקת האחרת.

מחלקות פנימיות

```

public class House {
    private String address;
}

public class Room {
    private double width;
    private double height;

    public String toString() {
        return "Room " + address;
    }
}

```

גישה לשדה פרטי של המחלקה העוטפת

כותרת: מחלקה פנימית גישה לשדות ולשירותים (כולל `private`) של המחלקה העוטפת, וכן ל`toString()`.

מחלקה פנימית (Inner Class) – מחלוקת שהוגדרה ב-`scope` של מחלוקת אחרת.

מחלקה החיצונית, אם וposite ייצור שדה של מסוג המחלוקת הפנימית יש לעשות זאת במדויק. הגדרת מחלוקת הפנימית מרמזת על היחס בין שתי המחלוקות: למחלוקת הפנימית יש שימושות רק בהקשר של החיצונית, יש לה היבנות אינטימית עם החיצונית, והיא מחלוקת עזר של החיצונית. כל מופע של עצם מטיפוס המחלוקת הפנימית, משוויר לעצם מטיפוס המחלוקת העוטפת, וכך:

- יש לחבר מיוחד לבני (בתרגום).
- על עצם מטיפוס המחלוקת הפנימית יש שדה הפניה שמיוצר אוטומטית לעצם מהמחלקה העוטפת.
- כותרת מוך יש למחלוקת הפנימית גישה לשדות ולשירותים (כולל `private`) של המחלקה העוטפת, וכך:

נקודות נוספת:

מחלקות פנימיות סטטיות – ניתן להגדיר מחלוקת פנימית כ-`static`, ובכך לציין שהיא אינה קשורה למופע מסוים של המחלוקת העוטפת. הדבר אנלוגי למחלוקת שבשירותה הוגדרו כ-`static` והוא משתמש בספריה עבור מחלוקת מסוימת. במקרה זה לא נוכל לעשות `return "Room" + address` והוא שדה מופיע (וביוון שהגדרכנו `static` לא **חייב להיות קיים אובייקט מטיפוס House בכל פעם שאנו מיצרים Room**, רק `h.address` מוקדם).

אם היינו למשם מקרים בפרט לפונקציה אובייקט `House`, יכולנו לרשום `h.address` – כיindein אפשר לגשת לשדות **private** (2021-8).

מחלקות פנימיות בתוך מתודות – ניתן להגדיר מחלוקת פנימית בתוך מתודה של המחלוקת החיצונית. הדבר מגביל את תחום ההכרה של אותה מחלוקת בתחום אותה המתודה בלבד. **מחלקה הפנימית תוכל להשתמש במשתנים מקומיים של המתודה, רק אם הם הוגדרו כ-final** (החול מ-8 Java ניתן לגשת גם למשתנים שמתנהגים כמו `final`, בלומר **מקבלים השמה פעם אחת בלבד לאורך חיים – אם בצע השמה יותר מפעם אחת נקבל שגיאה**:

(local variables referenced from an inner class must be final or effectively final

```

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

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

```

אם `toString` יכולה
לגשת גם ל-`num`?

מה יקרה לקובד לאחר
הוספה שורה זו?

מחלקות אונונימיות – בעדרת מחלוקות פנימיות ניתן להגדיר מחלוקות אונונימיות (לא שם). מחלוקות אונונימיות שימושיות מאוד במערכות מונחות אירושים וילמדו בהמשך הקורס.

הידור של **מחלקות פנימיות** – הקומpileר יוצר קובץ `.class` עבור כל מחלוקת. מחלוקת פנימית אינה שונה במובן זה ממחלוקת רגילה. שם המחלוקת הפנימית יהיה `Outer$Inner.class`.

- מחלקות פנימיות סטטיות – מחלוקת שהוגדרה ב-`scope` של מחלוקת אחרת.
- מחלקה החיצונית, אם וposite ייצור שדה של מסוג המחלוקת הפנימית יש לעשות זאת במדויק. הגדרת מחלוקת הפנימית מרמזת על היחס בין שתי המחלוקות: למחלוקת הפנימית יש שימושות רק בהקשר של החיצונית, יש לה היבנות אינטימית עם החיצונית, והיא מחלוקת עזר של החיצונית. כל מופע של עצם מטיפוס המחלוקת הפנימית, משוויר לעצם מטיפוס המחלוקת העוטפת, וכך:

- יש לחבר מיוחד לבני (בתרגום).
- על עצם מטיפוס המחלוקת הפנימית יש שדה הפניה שמיוצר אוטומטית לעצם מהמחלקה העוטפת.
- כותרת מוך יש למחלוקת הפנימית גישה לשדות ולשירותים (כולל `private`) של המחלקה העוטפת, וכך:



נחזיר ל-Cell myList. כדי להסתייר מהלוקה של הרשימה את הייצוג הפנימי, ובדי לאפשר גישה לשדות הפרטאים של Cell נכתוב את Cell כמחלקה פנימית בתוך myList. ישנן שתי אפשרויות:

1. **Cell פנימית לא סטטית** – כל עצם מסוג Cell משווין לעצם myList. במקרה זה, Cell לא חייבת להיות מוגדרת בוגריה. טיפוס התוכן של ה-Cell נקבע על פי הפרטער האקטואלי של עצם myList המתאים. הרשימה קובעת את סוג האיברים, וכל האיברים שותפים לו – **טיפוס הגנרי T**. לעומת Cell generic גם כן, אבל הפרטער שלו מוגדר **פושט**.

הערה: נראהות השדות והשירותים של מחלקה מקוננת פרטית אונה שימושותית (בכל מקרה הם ידועים רק למחלקה העוטפת).

2. **Cell פנימית סטטית** – נמצאת בתוך myList אך לא מחיבת קיום של אובייקט myList. ל-Cell פרטער גנרי משלו, head יכול להיות תוליה במחלקה myList, והפרטער הזה חייב להיות לא-S, ונסמכו-B-S. נשים לב שהשדה head שנמצא במחלקה myList, מוגדר עם הפרטער T. המחלקה Cell הפנימית מבצעת השמה לערך שנשלח עבורה הפרטער הפנימי שלו שנקרא S.

```
public class myList<T> {
```

```
    private static class Cell<S> {
        private S cont;
        private Cell<S> next;

        public Cell(S cont, Cell<S> next) {
            this.cont = cont;
            this.next = next;
        }

        public S cont() { return cont; }
        public Cell<S> next() { return next; }
        // ...
    }
}
```

```
private Cell<T> head;
private Cell<T> curr;
```

```
public class myList<T> {
```

```
    private class Cell {
        private T cont;
        private Cell next;

        public T cont() { return cont; }
        public Cell next() { return next; }
        // ...
    }

    private Cell head;
    private Cell curr;

    public myList(...) { ... }
}
```

איסטרטורים

נתבונן בפונקציה printList(). בעת הפקציה פונה למסך על ידי System.out.print, וזו החלטה שיש לשמר "למן קונפיגורציה". אפשר לחשב על שימוש בtoString(), שתחזיר את איברי הרשימה במחוזות. הבעה שהפונקציה מכתיבה את פורמט הדפסה.

■ אנחנו צריכים למחלקה myList **יהיה מושה שידוע להציג תשובה על שתי שאלות:**

1. מהו האיבר הבא?

2. האם נותרו איברים נוספים?

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

public interface Iterable<E>{
    Iterator<E> iterator();
}

public class myList<E> implements Iterable<E>{
    ...
    public Iterator<E> iterator() { // implementation }
}
```

הערה: הפונקציה next() קצת בעייתית, היא גם מחזירה את האיבר הנוכחי, וגם מקדמת את האיטרטור שיצבע לאיבר הבא. לעומת זאת מוצעת שתי פועלות מוגנון שונה בבת אחת.

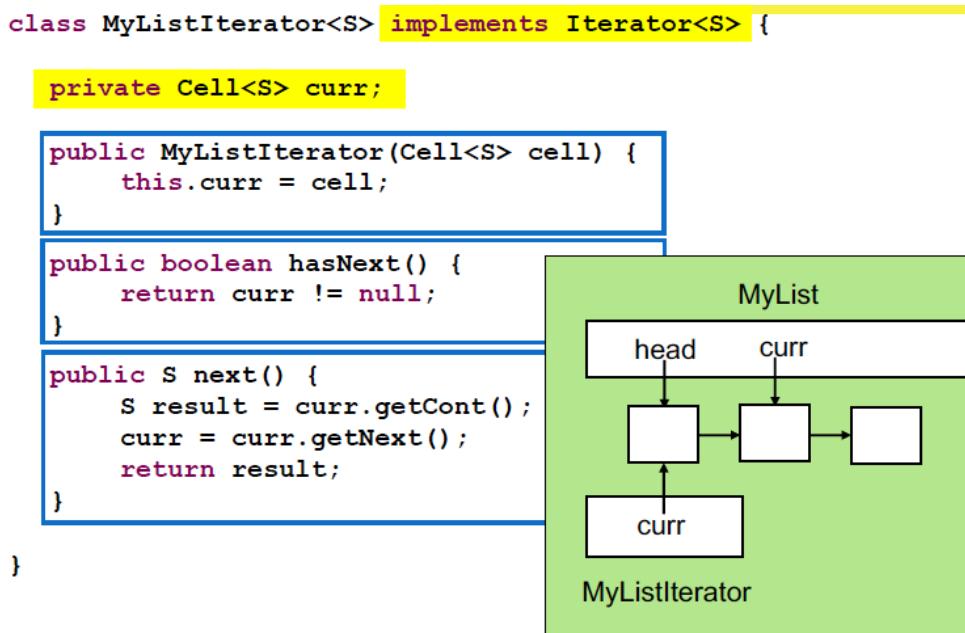
נשים לב כי הממשק Iterator מגדיר את טיפוס הנתונים איטרטור עצמו (**אני יודע להחזיר איבר**). המחלקה שלנו, כדי שהיא תהיה מוגדרת לאיטרטיה, היא צריכה למש את הממשק Iterable (**עליה אפשר לרוץ בוללאה**) שהוא מקשר בין האיטרטור עצמו. במנשך זה פונקציה שנקראת iterator שמחזירה אובייקט מטיפוס האיטרטור <E>, ועושים בכך שימוש בטיפוס גנרי E כמובן. כמו כן, המחלקה myList מימוש את הפונקציה iterator() של הממשק Iterable ומחזיר אובייקט שניון לבצע עליו את שתי השירותות הראשיות – ()next(), hasNext(). כיצד נழמץ את המחלקה שמייצגת את האובייקט הזה? ממש נכון.



באופן כללי, אם יש לנו אוסף בלשונו (**שummash at ha-mashuk** [Iterable]), אנחנו קוראים לפונקציה `iterator` שלו ומקבלים את האובייקט שמבצע איטרציה. האוסף אחראי לספק לךו איטרטור תייני (עצם מחלוקתה שsummash את המASHUK [Iterator]) המאותחל לתחלת מבנה הנתונים באוסף.



לבן, אם נרצה שהמחלקה `MyList` תספק ללקוחותיה את האפשרות לסרוק את כל האיבר ברישימה, علينا לכתוב לה `Iterator`. כמובן, אנו ניצור מחלוקת חדשה בשם `MyListIterator` שsummash את המASHUK [Iterator], ובעצם מייצגת איטרטור על איברי האוסף `.next`, `hasNext`. מחלוקת זו תsummash את כל הפונקציות של המASHUK [Iterator] ב�. `MyList`



בר הגדרכנו טיפוס חדש שהוא נוכל להחזיר בפונקציה `iterator()` – שמחזירה אובייקט מטיפוס `Iterator` (בפועל, אובייקט של מחלוקת summash את המASHUK [Iterator]). מחלוקת המsummash את המתוודה `curr` בעצם/mmsummash את המASHUK [Iterator] המוביל מתודה זו בלבד. הziמוד בין המחלוקת `MyList` והמחלקה `MyListIterator` חזק – על כן מקובל למשת את האיטרטור כמחלקה פנימית של האוסף שהוא פועל. בעת הלקוח יכול לבצע פעולות על כל איברי הרישימה בלי לדעת מהו המבנה הפנימי שלו – הוא מקבל אובייקט לעבוד איתו בצד יבצע איטרציה.

```

public class MyList<T> implements Iterable<T> {
    //...
    public Iterator<T> iterator() {
        return new MyListIterator<T>(head);
    }
}
  
```

הערה: אם לא היה לנו את `MyList`, `Iterator`, `Iterable` בעצמו, לא היינו יכולים ליצור איטרטורים שונים בעלי states שונים (במשמעות הנקחי שלנו), אפשר ליצור מאותו אובייקט `MyList` איטרטורים שונים, בעזרה קרייה לfonקציה (`iterator()`). היינו יכולים לבצע רק לולאה אחת של איטרציה ולא לולאה מקוננת: יש לנו אובייקט `MyList` אחד, ואם הוא עצמו האיטרטור אז יש לנו רק איטרטור אחד.

נקודות נוספות:

- ראינו את הפונקציה `printSquares` שמדפסה את ריבועי הרשימה באמצעות איטרטור, בלי להשתמש בעובדה שזו אכן רשימה. נשים לב כי בתבוננו (`i*i`)
`println(iter.next() * iter.next())`. יש פעמים שבן להעתיל ולא לשמר ערך חזרה של פונקציה במשתנה, יוצר בכך שקריה נוספת לפונקציה, לא תחשב את אותו הערך, אלא היא גם מקדמת את האיטרטור זהה בעיתוי. **יש לשמור על הפרדה בין פקודות לשאלות.** בנוסף, לשים לב שלפעמים כן כדאי **לשמר ערך חזרה מקרייה לפונקציה במשתנה.**
- על כל אוסף נתונים שמממש את המנשך `Iterable`, אפשר לבצע איטרציה ישירה באמצעות **ולאת for-each**.

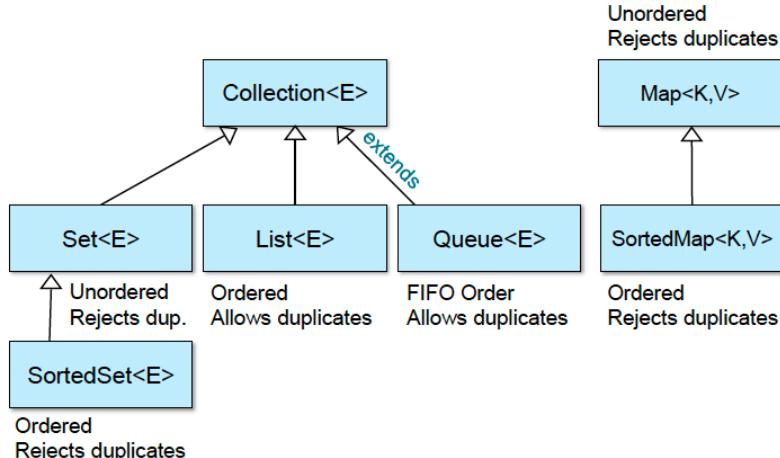
אוספים גנריים (תרגול 7)

מנשכים:

נסתכל תחילה על המנשכים של האוספים השונים:
 הם כוללים גנריים, ומחזקים טיפוסים שאינם פרימיטיביים – `Integer`, `String` וכו'. כל מנשך זה מייצג ADT מסוים. ישנים מימושים שונים אפשריים למנשך, ובעת נראה אילו מבני נתונים שונים אפשריים כדי למשבץ כל ADT.

מימושים (מחלקות):

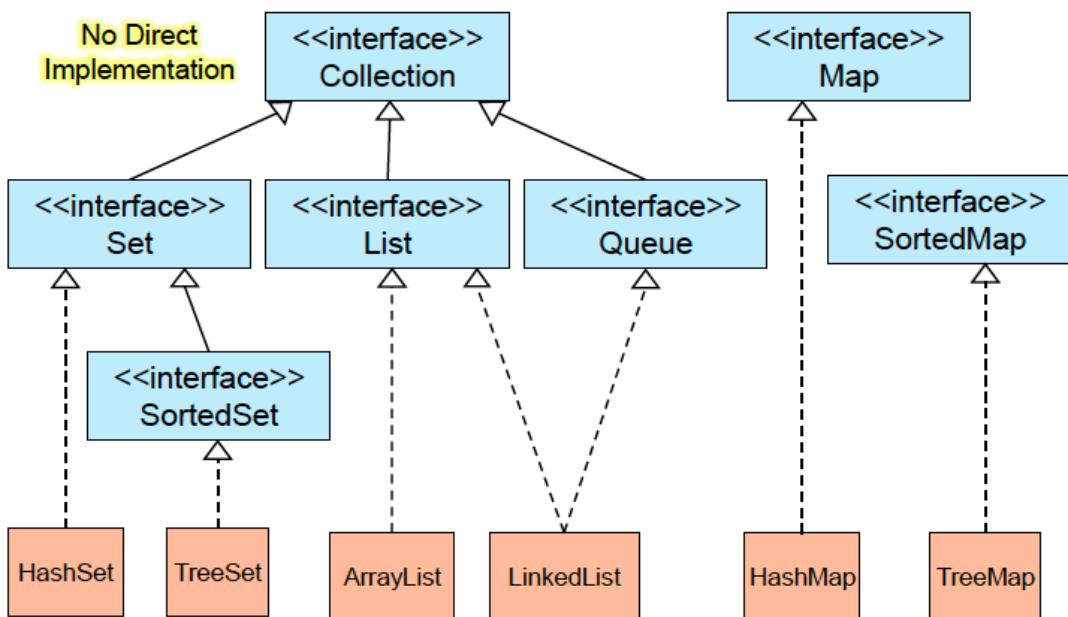
המבנה הנדרש לשמות המחלקות שמממשות את המנשכים: `<Data Structure> <Interface>` כולם **קודם כל**anno מצינים את מבנה הנתונים, ואז את **ה-ADT** שהוא ממש.



General Purpose Implementations		Data Structures			
		Hash Table	Resizable Array	Balanced Tree	Linked
Interfaces	Set	HashSet		TreeSet (SortedSet)	LinkedHashSet
	Queue		ArrayDeque		LinkedList
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap (SortedMap)	LinkedHashMap



נשים לב לתרשימים נוספים המגדיר את היחסים בין המחלקות. חוץ מל מסמל extends (ירושה), וחוץ מוקוקו מסמל implements (מימוש של המנשך):



Collection – מכיל אוסף של פעולות שימושיות כלילות לאוסף. יש פונקציה שמחזירה איטרטור. זה המנשך היחיד שאין לו מימוש ישיר (רק מנשכים שיורשים ממנו).

נסתכל על ה-ADTs (המנשכים) השונים וביצד הם מתנהגים:

ADT	סדר	כפליות	התנהגות
Map גישה לאיבר ב-(1) 0	אין	אסור	<p>מגדירים שני טיפוסים: key ו-value. אם מבצעים put לאותו key עם value אחר, זה מעודכן את ה-value של אותו key.</p> <p>אם נרצה לשמר על סדר אפשר להשתמש ב-LinkedHashMap.</p> <p>יש גם את ה-ADT SortedMap שהוא שומר על סדר. נרצה מצביע מטיפוס המנשך הנ"ל כדי להשתמש בפונקציות של המנשך SortedMap.</p>
SortedMap			<p>Map בבעצמו הוא לא Iterable. יש פונקציות שונות שמחזירות הסתכלות שונה על המפתחות שלו, על הערכים שלו, ועל צמדים של מפתחות וערכים (דווש לעובוד עם האובייקט Map.Entry איבר במילון הנ"ל, ומוביל פונקציות getValue(), getKey()):</p>
			Three views of a Map<K, V> as a collection
Set SortedSet	אין	אסור	<p>אין אפשרות איברים כפליים (אם שניהם הינהו שמתקיים $y.equals(y)$). הפונקציה remove יכולה לקבל רק אובייקט, כיון שאין משמעות לאינדקס באוסף שלא שומר על סדר. אם נרצה לשמר על סדר ספציפי אפשר להשתמש בימוש עם TreeSet, או ב-LinkedHashSet. בשימוש עם HashSet אין הבטחה של סדר.</p>
List	יש	ניתן	<p>הפונקציה remove יכולה לקבל אינדקס לאובייקט שרחצים למחוק, או את האובייקט הספציפי. יש בכך העמסה. אם נعتبر את המספר 3 מדובר בטיפוס פרימיטיבי וכן זה אינדקס.</p>
Queue	FIFO	אסור	<p>הפונקציה remove מסירה את האיבר הראשון שנכנס (הראשון בתור) אם הוא לא מקבלת ארגומנטים. האיברים מסודרים באופן לפי סדר ההכנסה.</p>

אלגוריתמים:

במחלקה `Collection` מוגדרים אלגוריתמים שונים: מיון, חיפוש, וכו'. למשל עבור מיוון נוכל לקרוא ל-(`Collections.sort(list)`). איך בזמנים אנחנו ממיינים את האיברים של רשיימה שמכילה איברים מטיפוס `T`?

1. **ללא Comparator** – דבר זה מחייב את איברי הרשימה למשמש את המנשך `<T>`. כלומר, המחלקה שמייצגת את טיפוס איברי הרשימה תמשש את הפונקציה `compareTo`. באופן טבעי, **מחלקה כמו `String` מימושת את המנשך Comparable<String>** וכאן ניתן לשולח רשיימה של מחזורות לפונקציה `sort`.

```
public class Point implements Comparable<Point>{
    ...
    public int compareTo(Point other) {
        //comparison by the x axis
        Integer.compare(this.x, other.x);
    }
}
```

- The program:

```
List<Point> pointList = new LinkedList<Point>();
pointList.add(new Point(1, 3));
pointList.add(new Point(0, 6));
Collections.sort(pointList);
System.out.println(pointList);
```

2. **עם Comparator** – אנחנו יוצרים מחלקה נפרדת שהיא מימושת את המנשך `Comparator<T>` ונעביר אותה כארוגמנט לפונקציית `sort`. **נשים לב שכיוון שאין מחלקה נפרדת, אין לה גישה ישירה לשדות של T**, בדוגמה כאן למשל אין לה גישה לשדות של `Point` וכן היא מביעת `() getY()`. **היתרון כאן – אפשר ליצור כמה Comparators שונים (מחלקות שונות), ולהשתמש בהם במקרים שונים, כך יוכל לבחור כיצד אנו רוצים למיין באמצעות השוואות שונות.** לעומת זאת `compareTo` רק דרך השוואת מיוון.

```
public class YAxisPointComparator implements Comparator<Point> {
    public int compare(Point p1, Point p2) {
        //comparison by the y axis
        return Integer.compare(p1.getY(), p2.getY());
    }
}
```

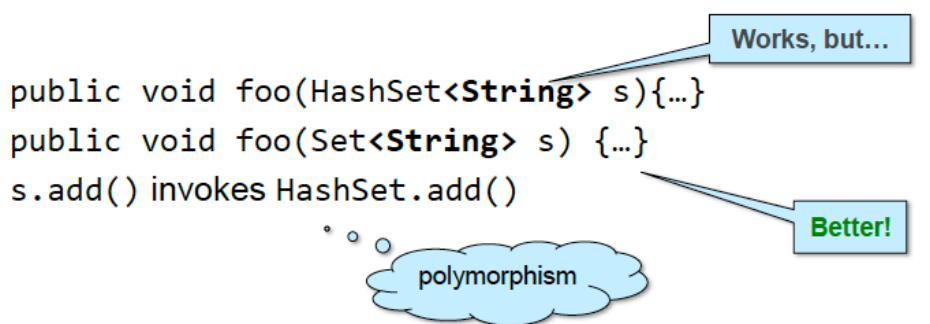
- The program:

```
List<Point> pointList = new LinkedList<Point>();
pointList.add(new Point(1, 3));
pointList.add(new Point(0, 6));
Collections.sort(pointList, new YAxisPointComparator());
System.out.println(pointList);
```

- The output: `[(1,3), (0,6)]`
- Useful for sorting existing classes (e.g., String)

נקודות חשובות:

- נרצה שהטיפוס של האובייקט שנגדי יהיה **מطيפוס המנשך**, והטיפוס הדינמי בפועל יהיה אחד המימושים שלו.凝דיף להשתמש **בטיפוס המנשך גם בפונקציות**, ולצין את המימוש הספציפי (מבנה הנתונים) רק בעת ייצורו האובייקט.

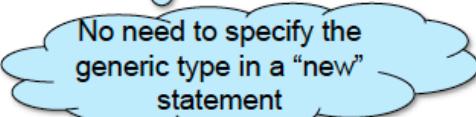


• Diamond Notation – כתיב חסר" בצד ימין שבו אנו יוצרים את האובייקט. יש השלמה אוטומטית של הטיפוס הכללי.

חייבים לשים את הסוגרים המשולשים עצם (אם לא, יהיה כאן Compilation warning, מדובר בטיפוס raw!)

```
Set<String> s = new HashSet<String>();
```

→ **Set<String> s = new HashSet<>();**



```
Map<String, List<String>> myMap =
    new HashMap<String, List<String>>();
```

→ **Map<String, List<String>> myMap = new HashMap<>();**

Not the same as:

```
Map<String, List<String>> myMap = new HashMap();
```

(Compilation warning)



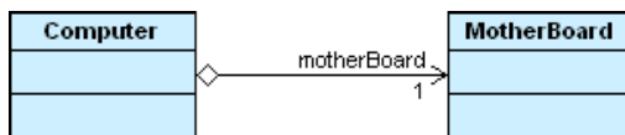
3 - ירושה

ירושה

יחסים בין מחלקות

בעיית המלבן: יש לנו מימוש של מלבן ורגיל. נרצה לבנות מחלוקת המייצגת מלבן צבוני. כיצד נוכל לעשות שימוש במלבן הרגיל לשם מימוש המלבן הצבוני? יצאנו לבדוק... נציג 3 גרסאות למחלוקת והחסרונות של כל גרסה. לבסוף, נתמקד בגרסה השלישייה ונחקור דרכה את מבנהו היורשה.

1. שכפול קוד – הקוד יהיה מאוד דומה לקוד של המחלוקת Rectangle שראינו, פרט לשינויים מאד קטנים. אין כאן הצדקה לשכפל את כל הקוד – זה תחזקה כפולה, תיקון באגים כפול, לא טוב.
 2. הכליה (aggregation) – המלבן הצבוני יכול להיות מלבן ורגיל. בנוספ, שדה של צבע. את רוב הפעולות של המלבן יבצע השדה שהוא המלבן הרגיל (האצלה/delegation). בעת קל יותר לתזק במקביל את שני המלבנים, כל שינוי בחלוקת Rectangle יתבטא אוטומטית בחלוקת ColoredRectangle (במיעט, פרט לפונקציונליות חדשה).
- ביחסים בין מחלקות, היחס הבסיסי הוא **Association** (היברות, קשרות, שיתופיות). מכאן מגדים:
- הכליה – מבטא הכליה, "has a". החלקים עשויים להתקיים גם ללא המיבל, המיבל מכיר את רכיביו אבל לא להיפך. בדרך כלל לאוסף ישיחס צזה עם רכיביו.

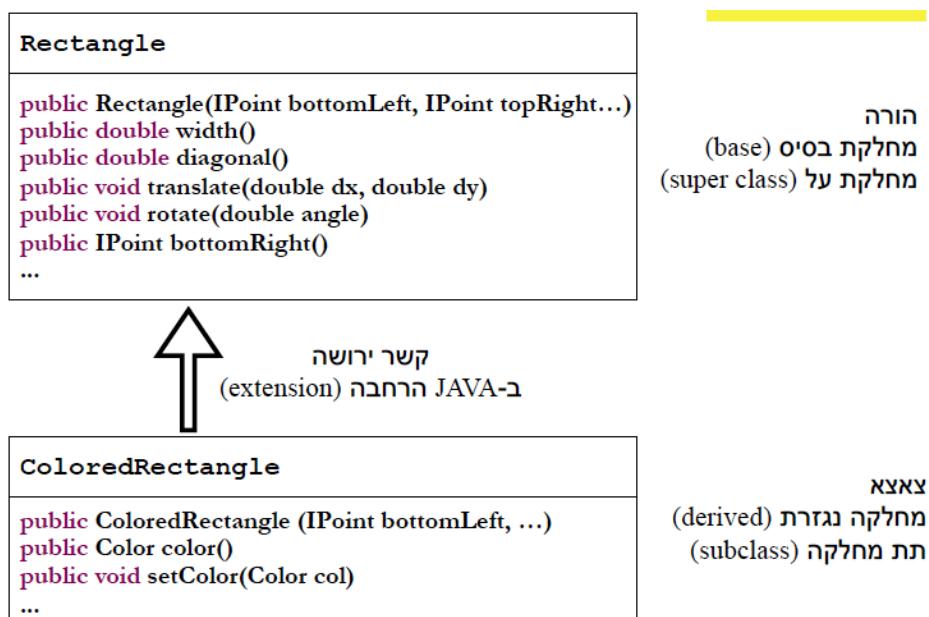


- Composition – דומה להכליה, אבל מבטאים יותר חזק של הרכבה, "part of". ראיינו שנitin לבטא הרכבה ע"י שימוש בשדה מופיע שטייפוסו הוא מחלוקת פנימית, אולי מקרה מאד קיצוני של הרכבה (עם תלות הדוקה בין המחלוקות).
- 3. ירושה – היחס שבין מלבן ונקודותיו הוא יחס של הכליה. לעומת זאת, היחס בין מלבן צבוני ומלבן (רגיל) הוא יחס של "סוג של". נבטא יחס זה באמצעות ירושה.

ירושה כיחס-a-is

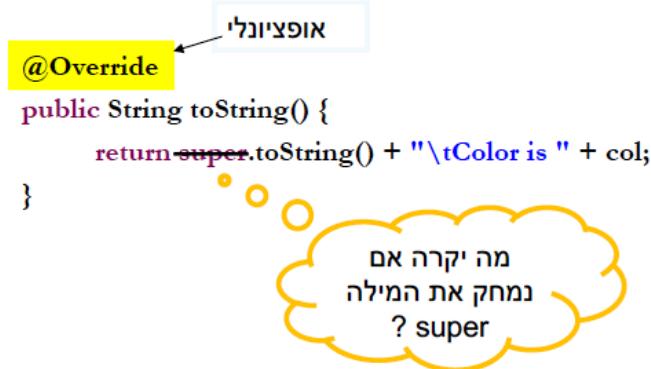
כאשר מחלוקת היא סוג של מחלוקת אחרת, אנו אומרים שהלמעלה היחס **a-is**. יחס זה נקרא גם **Generalization**. ניתן לראות במחלוקת החדשה מקרה פרטי, של המחלוקת המקורית. בדרך כלל יהיו למחלוקת תכונות ייחודיות, המאפיינות אותה, שלא באו לידי ביטוי במחלוקת המקורית (או שボוטאו בה בכללות).

מחלקה אשר תכרייז על עצמה **sheIs** מחלוקת אחרת, תקבל בירושה את כל תכונות אותה מחלוקת (במעט). כל מחלוקת ב-Java מרחיבה מחלוקת אחת בדיקון (מנשכים ניתן **למשמש במה שחוצים**). מדובר יחס זה לא דומה למה שעשינו עם מנשך? במנשך הפונקציות לא היו עם מימוש, בעוד אנחנו מרחיבים מחלוקות עם מימוש קיים, ומרחיבים את הפונקציונליות שלהן במרקם פרטי יותר.



תופעות בעולם הירושא:

- **בנאי** – מחלקות בנות מלמעלה למיטה, כלומר מההורה הקדמון ביותר. **השורה הראשונה בכל בניין** כוללת קריאה לבניין מחלקת הבסיס ע"י `super(constructorArgs)`. אם לא נכתבו עצמן את הקריאה לבניין מחלקת הבסיס, יוסיף הקומפיילר בעצמו את השורה `super()`, **ואם למחלקה הבסיס אין בניין ריק נקבע שגיאת קומפיילציה**.
- **אופציה** – נוספת היא קריאה לבניין אחר של המחלקה היורשת באמצעות `this` (ראינו בבר, תחביר זה לא מיוחד ליורשה). בעצם, זה לא סותר את הדרישה שהפעולה הראשונה שתתבצע בפועל היא קריאה לבניין של מחלקת הבסיס (כי בניין שאנן קוראים לו באמצעות `this`, תהיה קריאה ל-`super`).
- **הוספת שירותים** – המחלקה היורשת יכולה להוסיף מתודות נוספות, שלא הופיעו במחלקה הבסיס. **דרשת שירותים overriding** – מחלקה יכולה לדרס מודה שהיא יכולה בירושה – משיקולי ויעילות, או הוספת "תchromi אחירות", התבססות על שדות שייחודיים למחלקה היורשת. **על המחלקה היורשת להגדיר מודה בשם זהה ובחתימה זהה, למודה שהתקבלה בירושה (אחרת זהה הענסה ולא דרישת).** כדי להשתמש במתודה שנדרסה, ניתן להשתמש בתחביר הבא: `@Override @Override methodName(arguments)`. רצוי לציין `super.methodName(arguments)` מעל מודה שמתבצעת דרישת.



- אם לא נכתב `super` ותרתבצע קריאה עם `this`, קריאה אינטואיטיבית לאותה הפונקציה. האם שירותים סטטיים נורשים – כן, אבל זה לא נכון לקרוא מחלקה יורשת לפונקציה סטטית של מחלקת האם. בהמשך הקורס **נראה ששירותים סטטיים שנורשים מתנהגים בצורה שונה** משירותי מוףע שנורשים.
- **עקרון החלפה (substitution principle)** – בכל הקשר שבו משתמשים במחלקה המקורית, ניתן להשתמש במחלקה החדשה במקומה והקוד יעבד. נשתמש במנגנון היורשה רק כאשר המחלקה החדש **מחייבת יחס-is-a** עם מחלקת **קיימת וכן נשמר עקרון החלפה**.
- **פולימורפים וירושא** – כפי שאובייקט מטיפוס מנשך יכול להציגו לאובייקט מטיפוס שמנמש את המنشך, גם אובייקט ממחלקת מקורית יוכל להציגו לאובייקט ממחלקת יורשת מהמחלקה המקורית. כאן יש לשים לב להבדל בין טיפוס סטטי (סוג המשתנה) לפי הטיפוס הדינמי (מה שהוא מצבו אליו בפועל).

טיפוס סטטי ודינמי:

Rectangle r = new ColoredRectangle3(...);
Compile time type Runtime type

לABI הפניות (references) לעצמים אנו מבחינים בין:

1. **טיפוס סטטי (זמן קומפיילציה)** – הטיפוס שהוגדר בהכרזה על הפניה (מנשך או מחלקה). הקומפיילר הוא סטטי, והפעלת שירות על הפניה **חייב את הגדרת השירות בטיפוס הסטטי של הפניה**. טיפוס סטטי של משתנה צריך להיות הכללי ביטור האפשרי בהקשר שבו הוא מופיע.
2. **טיפוס דינמי (זמן ריצה)** – טיפוס העצם המוצבע בפועל, שחייב להיות נגזרת של הטיפוס הסטטי. מנגנון זמן הריצה הוא דינמי, פוליפורמי, והשירות שיעפע הוא השירות שהוגדר בעצם המוצבע בפועל (הטיפוס הדינמי של הפניה).



```

void expectRectangle(Rectangle r);
void expectColoredRectangle(ColoredRectangle3 cr);

void bar() {
    Rectangle r = new Rectangle(...);
    ColoredRectangle3 cr = new
    ColoredRectangle3(...);

     r = cr; —————
     expectColoredRectangle(cr);
     expectRectangle(cr);
     expectRectangle(r);
     expectColoredRectangle(r);
}

```

הטיפוס **靜態** של `r` נשאר
.Rectangle
הטיפוס **динامي** של `r` הופך
.ColoredRectangle3 להיות3

למרות שהטיפוס **динامي** של `r` הוא
ColoredRectangle3, אנחנו מקבלים שגיית קומפליציה

נראות וירושה

מחלקה ירושת לא יכולה לגשת לתכונותיה הפרטיות (`private`) של מחלקת הבסיס ישירות, אלא רק באמצעות מתודות ציבorias (`public` שהוא מודול מסורבל). לשם כך הוגדרה דרגת נראות חדשה – `protected`. **שודות שהוגדרו protected** מאפשרים גישה מותן: המחלקה המגדירה, מחלקות נגזרות (ירושות), מחלקות באותו החבילה. אם נגדיר את השודות של `IPoint` כ-`protected`, המחלקות היורשות יוכל לגשת אליהם ישירות בלי בעיה.

Modifier:	Accessed by class where member is defined	Accessed by Package Members	Accessed by Sub-classes	Accessed by all other classes
Private	Yes	No	No	No
Package (default)	Yes	Yes	No (unless sub-class happens to be in same package)	No
Protected	Yes	Yes	Yes (even if sub-class & super-class are in different packages)	No
Public	Yes	Yes	Yes	Yes

מדוע בן להשתמש ב-`private`? יצא עם עודף כוח להפר את חוזה מחלקת הבסיס, להעביר את עצמו לлокוח המוצהפת לקבל את אביו ולשבור את התוכנה.

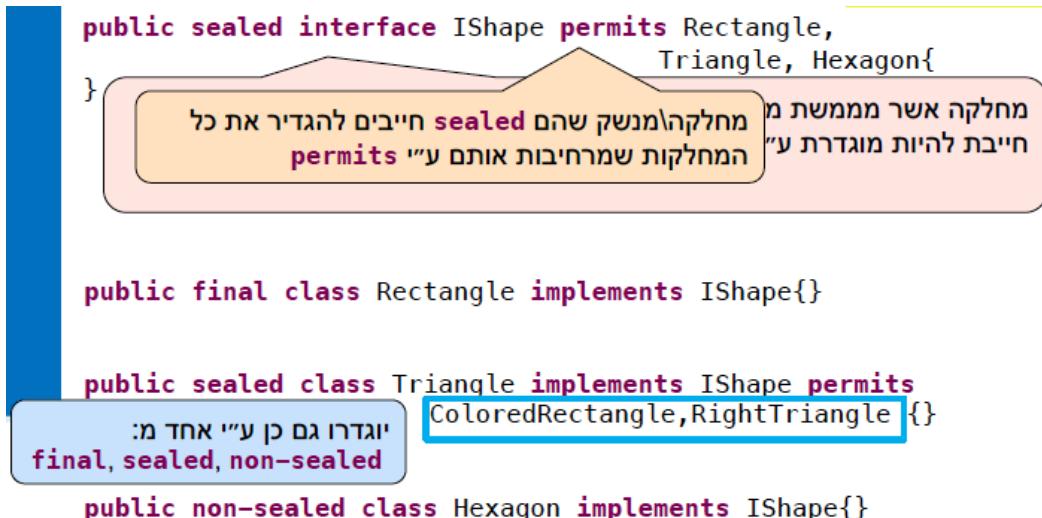
מכונית ירושה:

1. מתחודה שהוגדרה כ-`final` לא ניתנת לדרישת מחלקות נגזרות.
2. מחלוקת שהוגדרה כ-`final` לא ניתנת לירושה.

הגבלת באמצעות sealed

משהו קצת יותר רך.אפשר למסרים או מחלקות **לקבוע מי המחלקות שמרחיבות אותם**. לפני שהו sealed classes, ניתן היה למונע יוזה לחולון (final) או להשתמש במבנה שהם sealed/package/private על מנת להגביל את המחלקות המרחיבות.

- נשים לב להגדיר על ידי permits.
- מחלוקת אשר יורשת/מממשת מחלוקת שהוא sealed/manusk שחייב להיות מוגדרת ע"י אחד מבין האפשרויות הבאות: .final, sealed, non-sealed

**מחלקה Object**

כל מחלוקת ב-Java יורשת מחלוקת אחת בדיק. אם במחלוקת אנו לא כתבים extends, בירית המחלוקת היא extends Object. מהו בסיס כל המחלוקות, ומכללה מספר שירותים בסיסיים שככל מחלוקת צריכה. חלק מהמתודות קשורות לתכונות multithreaded וילמדו בקורסים מתקדמים.

פעם הייתה התייחסות ל finalize והיו על זה שאלות בבחינה, אך אין שם ש Kopiyot נספנות היום בנושא.

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
String	toString() Returns a string representation of the object.



ירושה II

מנשכים - המשך

איטרטור:

Interface Iterable<T>

Type Parameters:

T - the type of elements returned by the iterator

Iterator<T>

iterator()

Returns an iterator over elements of type T.

Interface Iterator<E>

Type Parameters:

E - the type of elements returned by this iterator

boolean

hasNext()

Returns true if the iteration has more elements.

E

next()

Returns the next element in the iteration.

default void

remove()

Removes from the underlying collection the last element returned by this iterator (optional operation).

1. הממשק **Iterable** – מתאר את האובייקט עליו נרצה לעבור בוללה (לרוב מדובר באוסף כלשהו). משמעו: ניתן לבצע על אובייקט זה מעבר באמצעות לולאת `for-each`. הממשק **Iterable** מכיר את הממשק **Iterator** ומחייב להשתמש בו (הממשק **Iterable** מגדיר פונקציה שמחזירה אובייקט מטיפוס **Iterator**).

2. הממשק **Iterator** – מתאר אובייקט שהוא מהוסף עליי נרצה לעבור בוללה. **כל אוסף** ניתן למendir מספר איטרטורים, כל אחד יעבור בסדר אחר או בחוויות אחרות על האיברים באוסף. בפרט, ניתן לבנות Iterator לאובייקט שאינו implements Iterable.

ראינו כיצד למש איטרטור עבור המחלקה `StackOfInts` – זמן – StackOfInts :

כדי שנוכל לבצע **for-each** על אובייקט מהמחלקה, היא צריכה למש את הממשק **Iterable**, ובמונן נדרש למendir טיפוס `Iterator` (אם לא נבצע `for-each`, יוכל לעבוד רק עם `Iterator` ולבצע את הוללה ישירות באמצעות `(while(it.hasNext())`).

נשים לב כי למחלקה `StackOfInts` אין צורך בפרמטר גנרי (מדובר במחסנית של שלמים) ולכן נדרש למendir את המחלקה בצהורה `implements Iterable<Integer>`.

איזו אינפורמציה האיטרטור שニצוץ, צריך לקבל בשבייל עבור כל האיברים במחסנית? את מערך האיברים במחסנית, אינדקס של התא האחרון במערך שמייצג מחסנית. איבר נוסף נשמר בשדה באיטרטור (`currIndex`), הוא התא הנוכחי שבו אנחנו נמצאים. נאותל את `currIndex` להיות 0 ונעדק אותו בהתאם לביצוע האיטרציה.

בונגע למחלקה `MyList`:

1. מה `MyList` צריך לדעת כדי לבצע איטרציה על `MyList`? את `head` שהוא מצביע לאיבר הראשון ברשימה. `MyListIterator` מוגדר במחלקה פנימית בתוך `MyList`, ובוון שלא מדובר במחלקה סטטית, אין צורך בפרמטר הגנרי.

הוא מכיר את T מהמחלקה `MyList`.

3. טיפול באיבר האחרון – באשר לנו מצביעים לאיבר האחרון, `hasNext` צריך להחזיר `True`, וכן `curr != null` ולא `null = curr.next`, `curr.next`, אחרית היום מפספסים את האיבר האחרון.

4. האם אפשר להסתדר בעלי השדה `zzz` (כיוון שגם מחלקה פנימית)? לא, אנחנו צריכים משתנה כלשהו לשימור על ה-`state` של האיטרטור.

5. האם אפשר להסתדר בעלי הארגומנט `head` בבנייה? כן, יש גישה לשדה ה-`private` של `MyList`.

**מנשקיים נוספים:**

המנשך **<T>** מתאר אובייקטים אשר משמשים להשוואת אובייקטים מטיפוס **T** האחד לשני.

Interface Comparator<T>**Type Parameters:**

T - the type of objects that may be compared by this comparator

int

compare(**T** o1, **T** o2)

Compares its two arguments for order.

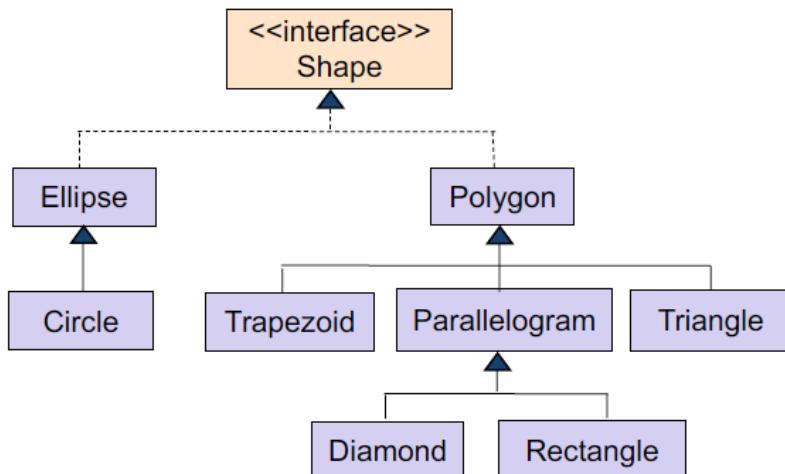
בדוק במו במקורה של **Iterator** ו-**Iterable**, מנשך אחד מתאר את האובייקט עצמו ("בר השווהה"), והשוו מאפשר להגדיר מחלקות שיכולות להשוות בין עצמים לפי קriterיוונים שונים. בשונה מ-**Iterable**-**Iterator**, המנשקיים **Comparable** (אותו) אפשר להשוות למשהו אחר (**Comparator**) (אני יודע להשוות בין שני דברים), אינם משתמשים האחד בשני.

מנשקיים וירושה:

1. בשם ששתי מחלקות מקיימותיחס ירושה, כך גם שני מנשקיים יכולים לקיים את אותו היחס. **מנשך, לעומת זאת מחלקה רגילה, כן יכול לרשות מספר מנשקיים שונים.** מחלוקת המממשת מנשך מחייבת למשוך את כל המתודות של אותו מנשך, **וכל המתודות שהוגדרו בהוריו של המנשך.**
2. אנחנו מפרידים בין **collection** ו-**map**:
 - o **map** אינו אוסף, ומכיל מיפוי בין זוגות. בולם הוא מקבל **שני טיפוסים גנריים**.
 - o הוא לא **Iterable**. לכן לא ניתן לזרע על **map** בולולה (אפשר לזרע או על ה-**keys** או על ה-**values**, או על-**items**).

מחלקות מופשטות

בهرרכיה המחלקות והמנשקיים שלנו, נגדיר מנשך **Shape**, שגם ימשכו שתי המחלקות Ellipse, Polygon, Shape.



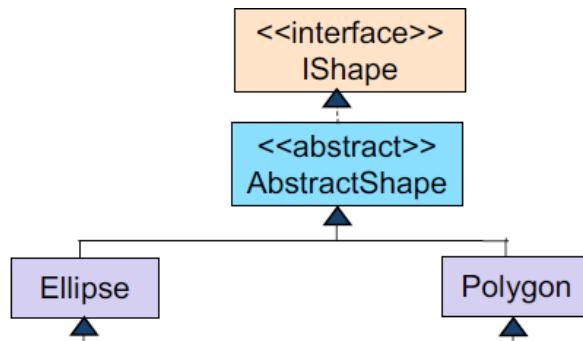
למצולע ולאליפסה יש שדה משותף שהוא צבע. עץ הירושה הב"ל, יגרום להגדרת השדה פערמים לשכפול קוד. מחד, לא ניתן להוסיף למנסך שדות או מימושי מתודות. מאידך, אם ייצור לשתי המחלקות מחלוקת שהיא אב משותף, דרך חישוב ההיקף תהיה שונה עבור כל מחלוקת. לשם כך קיימת המחלוקת המופשטת (abstract class) – **מחלקה עם מימוש חלקית**.

מחלקה מופשטת דומה למחלוקת רגילה עם הסיגרים הבאים:

1. ניתן לא למשוך מתודות שהגינו בירושה מחלוקת בסיס או מנשקיים (האחריות על המימוש עוברת למחלוקת שתירש את המחלוקת המופשטת).
2. ניתן להזכיר על מתודות חדשות ולא לממשן – כי כל מחלוקת תמשיך את המתודה בזורה אחרת.
3. לא ניתן ליצור מפעעים של מחלוקת מופשטת (למרות שהבנייה הבסיסי של המחלוקת המופשטת הוא אכן אפשרי שיתוף קוד בסיסי).
4. ניתן לממש מתודות – שימוש עיקרי כדי לחסוך בשכפול קוד.
5. ניתן להגדיר שדות – גם חוסך שכפול קוד לשדות שהם זמינים בהרבה מחלוקות.



מחלקות מופשטות משמשות בסיס ממשות למחלקות יורשות לצורכי חישוב בשכפול קוד. נגדיר את המחלקה `AbstractShape` המשמשת מושג `IShape`.
אות הממשק `IShape`.



מחלקה המופשטת `public abstract class AbstractShape implements IShape` מימושה רק חלק מן המתוודות של הממשק, כדי לחסוך שכפול קוד ביסוד היררכיה.

1. את המתוודות הללו מומומשות היא מצינית ב-`abstract`. אפשר לוותר על ההצעה של מתוודות לא מומומשות.
2. המחלקה שתירוש `IShape` צריכה למשוך את המתוודות של `IShape` שהיא לא מיישה.
3. ניתן ורצוי להגדיר בנאים במחלקה מופשטת, על אף שלא ניתן ליצור מופעים של המחלקה, הבניי ייקרא בעתיד מתוך בניאים של המחלקות היורשות (קריאות `super`) ויחסכו בשכפול קוד בין המחלקות היורשות.

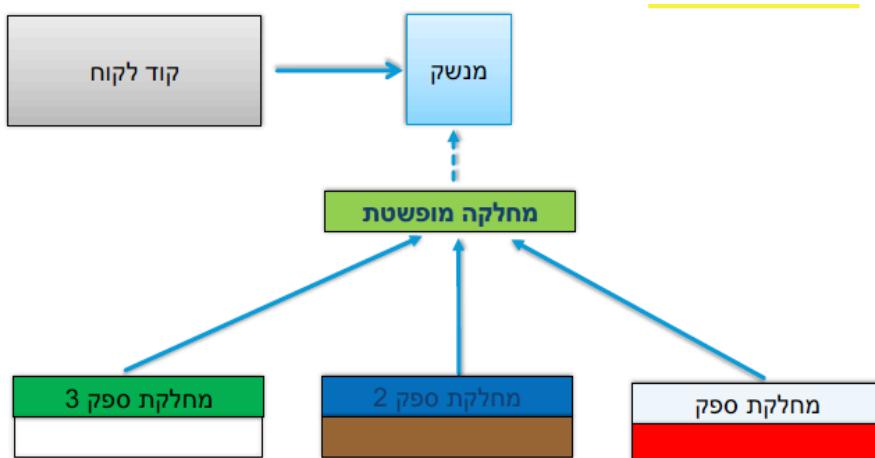
מנشك מול מחלקה מופשטת – על פניו יש עדיפות לשימוש במנשים ונתווות default (נתווות מופיע מומומשות בתוך הממשק).
לעומת ירושה, אין הגבלה על מספר המנשיים שאוטם מחלקה יכולה למסח.

- אולם, למחלקה מופשטת יש שדות, ניתן להגדיר בנאים וכן מתוודות שימושísticas בשדות.
- יתרו נוסף למחלקה מופשטת, ניתן למשוך מתוודות בדרגות נראות שונות (כאשר במנשך הנראות מוגבלת אך ורק ל-`public` ול-`private`). עם זאת, **נשים לב כי לא ניתן להגדיר פונקציות אבסטרקטיות private** במחלקה מופשטת.

גם במנשיים אנחנו קצת מוגבלים: **שימוש שני** במנשיים עם אותה פונקציה – אם מחלקה מימושה שני מנשיים (`I1`, `I2`) כאשר **שםם מימשו פונקציית default באוטם השם**, יש התנגשות בין שני המימושים. לכן, המחלקה חייבת לפתור את העמימות בכך שתתmesh בעצמה את הפונקציה. במשמעותם של המנשיים, או להתעלם מהם כלותן.

מחלקות מופשטות ומנסקים:

1. כאשר מגדירים מנשך ניתן לקבל את תהליך הפיתוח – צוות שימוש את המנשך במקביל לצוות שישתמש במנשך.
2. קוד לקו שנכתב לعباد עם מנשך בלבד ימשיך לשזהו ימשיך לרוץ גם אם יועבר לו ארגומנט עצם מחלקה חדשה המימושת אותו המנשך.
3. כאשר מחלקה מימושת מנשך אחד או יותר, היא נבנית מכל פונקציות השירות אשר כבר נכתבו עבור אותו מנסקים.





טיפוסי זמן ריצה

בשל הפלימורפיים אנו לא יודעים מה הטיפוס המדוק שאל עצמים – הטיפוס הדינמי עשוי להיות שונה שונה מהטיפוס הסטטי. בהינתן הטיפוס הדינמי, עשויות להיות פעולות נוספת שניתן לבצע על העצם המוצבע (פעולות שלא הוגדרו בטיפוס הסטטי). כדי לא להפעיל פעולות אלו علينا לבצע המרת טיפוסים (Casting) על הפינה.

ביצוע המרת – המרת טיפוסים נעזרת באמצעות אופרטור אונארי שנקרא Cast ונוסר על ידי כתיבת סוגרים מסביב לשם הטיפוס אליו רצים להמיר: <Expression> (Type). הדיוון באן אינו מתייחס לטיפוסים פרימיטיביים – רק לטיפוסי הפניה. הוא מייצר ייחוס מטיפוס Type עבור העצם שהביטו <Expression> מחשב, אם העצם **מתאים לטיפוס**:

1. **המרה למטה (downcast)** – המרה של ייחוס לטיפוס פחות כללי, כלומר הטיפוס Type הוא צאצא של הטיפוס הסטטי של העצם. המרה עלולה להיכשל, אם בזמן ריצה טיפוס העצם המוצבע לא תואם לטיפוס Type התוכנית עמוק (יזרkJ. ClassCastException).
2. **המרה למעלה (upcast)** – המרה של ייחוס לטיפוס יותר כללי (מחלקה או מנשך). המרה למעלה תמיד מצליחה, ובדרך כלל לא מצריכה אופרטור מפורש, היא פשוט גורמת לקומpileר לאבד מידע.
3. **כל המירה אחרת גוררת שגיאת קומPILEZA.**

לפניהם נרצה לבצע בדיקה, שהטיפוס אכן מתאים להמרה. נזכיר כי ההמרה אינה מסירה או מוסיפה שדות לעצם המוצבע. בזמן קומPILEZA נבדוק כי ההסתבה אפשרית (compatible types):

1. דרך אחת לבצע זאת היא "המתודה getClass Object המוגדרת ב- getClass()" – והשדה הסטטי class הקיים בכל מחלקה.
2. דרך שנייה היא "operator instanceof" – בודק האם הפניה a-is מחלקה כלשהי, כלומר האם היא מטיפוס אותה המחלקה, או יורשתה, או מממשה.

Pattern Matching

יש לנו שני records שמשם ממשים את הממשק Shape. בימוש הפונקציה getPerimeter() במנשך, מתחכמת בדיקה פעמיים של טיפוס האובייקט כדי לחשב בצוරה המתאימה אליו (הערה: **הדרך הנכונה McLatchile היא להגדיר את הפונקציה במופשטת** ושכל מחלקה תמשח אותה בנפרה).

כדי לקצר את התחריר של הגדרת ההמרה לאחר בדיקת instanceof, אנו עושים את זה בשורה אחת!

```
public interface Shape {
    public static double getPerimeter(Shape shape)
        throws IllegalArgumentException {
        if (shape instanceof Rectangle r) {
            return 2 * r.length() + 2 * r.width();
        }
        else if (shape instanceof Circle c) {
            return 2 * c.radius() * Math.PI;
        }
        else {
            throw new IllegalArgumentException("Unrecognized shape");
        }
    }
}

public record Rectangle(float length, float width) implements Shape {}

public record Circle(double radius) implements Shape {}
```

תבחר ממועד החל מ Java 17!

אפשר לשלב את זה בביטויים בוליאניים. זה עובד כיון שמדוברים בביטוי השני – > ()>().length()>5. רק כאשר החלק הראשון מתקיים. זה עובד עם האופרטור && ("וגם"). זה לא יעבוד עם "או", כי גם אם הראושן לא מתקיים הוא יבודק את התנאי השני.

```
if (shape instanceof Rectangle r && r.length() > 5) {
    // ...
}
```



דוגמיה:

הלקוח מקבל ארגומנט צורה גיאומטרית, ומנסה להפעיל פונקציית סיבוב. בדוגמה זו, לא הוגדר שירות סיבוב במחלקה Shape (גם לא שירות מופשט). מכיוון שלכל צורה שירות סיבוב שונה, על הלוקוח לבדוק את טיפוס העצם שהושבר לו בפועל ולבצע המרה בהתאם. הבדיקה מתבצעת באמצעות instanceof.

```
void rotate(IShape s, double degree) {
    if (s instanceof Polygon) {
        Polygon p = (Polygon)s;
        p.rotatePolygon(degree);
        return;
    }
    if (s instanceof Ellipse) {
        Ellipse e = (Ellipse)s;
        e.rotateEllipse(degree);
        return;
    }
    assert false : "Error: Unknown Shape Type";
}
```

מחלקות Polygon ו-Ellipse מממשות כל אחת פונקציה אחרת לסיבוב

כדי לשפר את הקוד ולהפוך אותו ליוטר Object Oriented, נשתמש במחלקה המופשטת AbstractShape אשר מספק ממשק אחיד לעובדה עם כל מי שיורש ממנה בהיררכיה.

```
abstract class AbstractShape implements IShape {
    //...
    abstract void rotate(double degree);
}

class Polygon extends AbstractShape {
    //...
    void rotate(double degree) {
        rotatePolygon(degree);
    }
}

class Ellipse extends AbstractShape {
    //...
    void rotate(double degree) {
        rotateEllipse(degree);
    }
}
```

בעת נוכל פשוט לקרוא לפונקציה rotate ובאמצעות הפולימורפיזם תיקרא הפונקציה המתאימה לפי העצם המוצבע בפועל. נותר רק לבדוק האם האם IShape הוא instanceof של AbstractShape (אין צורך לבדוק עבור כל מחלקה יורשת בנפרד), להמיר אותו כלפי מטה (עם התחבר המיחוץ), ואז לקרוא לפונקציה. הפונקציה מומשה במחלקות Polygon ו-Ellipse.

נשים לב לת לחבר המיחוץ: S הוא בבר אובייקט שהוא מטיפוס AbstractShape בשורה זו. אין צורך אחר כך לבצע casting.

```
void rotate(IShape s, double degree) {
    if (s instanceof AbstractShape as) {
        as.rotate(degree);
        return;
    }
    assert false : "Error: Unknown Shape Type";
}
```



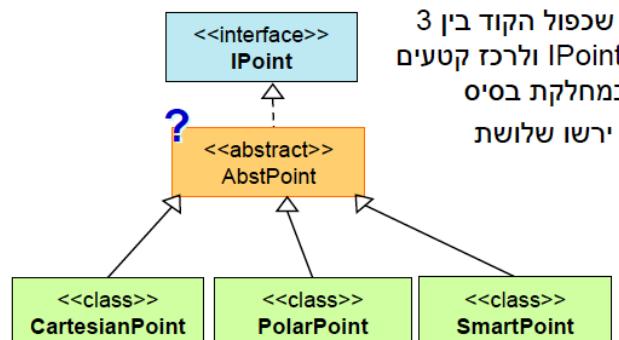
הורשה, מחלקות מופשטות, חריגים (תרגול 8)

מחלקות מופשטות:

- מחלקה מופשטת תמשח את המנשך, ותהייה מחלוקת ביןיהם. כל מחלוקת שemmמשת נקודה מסוימת, תבצע ירושה מהמחלקה המופשטת.

מנגנון ההורשה חוסך שכפול קוד בצד הספק

- ע"י ההורשה מקבלת מחלוקת את קטע הקוד בירושה במקום לחזור עליו. שני הספקים חולקים אותו הקוד



- ננסה להזורה את שכפול הקוד בין 3ימושי המנשך IPoint ולרכץ קטעים משותפים אלה במחלוקת בסיס משותפת ממנה ירשו שלושת המימושים.

מנגן זה אפשר לנו **למנוע שכפול קוד במחלקות היורשות**. נרכז את הקוד המשותף במחלקה בסיס שהיא מייצגת "נקודת מופשטת". נשים לב כי לא ניתן ליצור עצם מחלוקת מופשטת.

- ראיינו עברו Cartesian ועבור Polar, שיש להן **לוגיקה משותפת!** וכן העברנו את distance וגם את toString למחלוקת. אנו קוראים לפונקציה () getX() שהמחלקה שתריש תמשח, במחלוקת המופשטת פונקציה זו אינה ממומשת! **מחלקה המופשטת אינה חייבת למשח את כל המתודות במנשך – היא יכולה להשאיר חלק למחלוקות שירשו ממנה.**
- נזכר ב-sealed, כך אנו מגבילים את הירשה, ומגדירים בבדיקה מי יכול לרשות.

```
public sealed interface IPoint permits AbstPoint {...}
```

```
public abstract sealed class AbstPoint implements Ipoint permits PolarPoint,
CartesianPoint, SmartPoint {...}
```

```
public sealed class CartesianPoint extends AbstPoint permits angle{...}
```

```
public final class PolarPoint extends AbstPoint {...}
```

```
public non-sealed class SmartPoint extends AbstPoint {...}
```

ניתן להרחבה ללא sealed

חריגים:

- נרצה להשתמש בחיריגים כדי להתריע על שגיאות מתאימות בקוד: throw new Exception(). אנו יוצרים אובייקט מטיפוס Exception. אם אנו זורקים שגיאה אנו חייבים להוסיף להוספה בחתימה שהפונקציה זורקת שגיאה.

- נפריד בין Checked-to-Unchecked. במקרה של Checked:
 - אפשר לא לטפל בחיריג, ורק להציגו עליו.
 - אפשר לטפל בחיריג באמצעות try-catch.

- אם נוסיף catch כלשהו שתחזק exception, אז אנחנו מטפלים בכל שגיאה אפשרית ויכולים להתעלם משלויות שימוש על באג אפשרי! נוכל ליצור טיפוס חריג חדש לשימוש שלנו. **נמשח מחלוקת חדשה שיורשת מ-Exception, ונזרוק שגיאה מסוג המחלוקת הזאת במקום המתאים.**



חריגים

מבוא לחריגים

חריגים מבטאים מצבים יוצאי דופן, מקרי קצה ומצבים בלתי צפויים בリストת התוכנית בוגן: ארגומנטים שאיןם חוקיים, בעיות ברשף התקשרות, קובץ שאינו קיים ועוד.

:Hands on

1. כדי לזרוק חריג נכתב (...).throw new Exception(...)
2. אם נגדיר שירות המצהיר על חריג (נקتبו throws Exception), הקוד לא יתאפשר לשנקר לפונקציה ב-main אם לא נבחר באחד משני פתרונות:
 - o גם **main** תצהיר על חריג.
 - o **נתפל בחריג** – באמצעות בлок של try/catch, "נתפס" את השגיאה, ואז השגיאה נעלמת.

סוגים של חריגים:

1. **Checked exceptions** – תנאים אשר עשויים להתקיים במהלך ריצה תקין של תוכנית תקינה (מקרי קצה). תנאים אלו מיוצגים ע"י המחלקה **Exception** (חריגים שאיןם **RuntimeException**).
2. **Unchecked exceptions** – בעיות חמורות מיוצגות ע"י המחלקה **Error** וגם שגיאות בתוכנית מיוצגות ע"י המחלקה **RuntimeException**.

הקשר בין חריגים לחודים וליחס ספק-לקוח

החוויות שהגדכנו אינם סיימטריים, אם הלקוון אינו מצליח לקיים את תנאי הקדם, אין לו טעם בכלל לקרוא לשירות. אם הספק אינו מצליח לקיים את תנאי האחר, אין לו אפשרות לבטל את הקריאה לשירות, היא בבר התבכשה. הספק אינו יכול לבטל את העסקה. נוכל להגיד לנו תנאי קדם חלש יותר. לבר ייש כמה סיבות:

1. **חוסר שליטה** – בניית גישה לקובץ, יכול להיות שהוא נמוך בדיקות לפני שניגשנו אליו. כדי לוודא שהקובץ אכן קיים, לא משנה כמה או כמה לפניו, נצטרך להיעדר במנגנון אחר – חריגים.
 2. **קשה לבדוק את תנאי הקדם** – בפתרון מערכת משווהות באמצעות מטריצה, בדיקה האם היא הפיכה יקרה בערך כמו פתרון המערכת עצמה. עדיף לבצע לנטוטות לפתרור את המערכת, ושידוע לנו אם הוא נכשל כי המטריצה אינה הפיכה.
 3. **"הגורם האנושי"** – נתונה תוכנית המקבלת קלט מהמשתמש של מספרים שלמים ומדפיסת את הסכום שלהם. אמןם, שום דברינו מחייב שරקלט שהמשתמש מוזן תמיד יהיה תקין – בניית בצע (arg) – Integer.parseInt(arg).
- במקרה זה, **לפונקציה אין תנאי קדם שתמיד מתקיים**, ויש **טיפול בקלט מסויימים ע"י זריקת חריג**. הלקוון אינו מחויב לקיים את תנאי הצד (side condition), שימוש "נתיב מילוט" לספק. הדבר שונה מהגדלת תנאי קדם, שב
השירות מחייב שתנאי הקדם מתקיים ומתעלם ממקרים שבהם הוא אינו מתקיים.

```
public class AddArguments2 {
    public static void main(String args[]) {
        int sum = 0;
        try{
            for (String arg : args) {
                System.out.println("parsing: " + arg);
                sum += Integer.parseInt(arg);
            }
        }
        catch (NumberFormatException exp){
            System.out.println("one of the arguments is not an integer");
        }
        System.out.println("Sum = " + sum);
    }
}

> java AddArguments2 1 two 3.0 4
parsing: 1
parsing: two
one of the arguments is not an integer
Sum = 1
```

טיפול בחיריגטם – חריג יכול להזירק ע"י פקודת `throw`. קטע קוד אשר עלול לזרוק חריג יעתוף ע"י הלוקה בבלוק `try`. פקודת `throw` גורמת להפסקת הביצוע הרגיל, והמשערך מփש exception handler (בלוק `catch`) שיתפוס את החירג.

1. אם בלוק `catch` העוטף מביל טיפול בחיריג זה – **קטע הטיפול מתבצע, ולאחריו עוברים לבצע את הקוד שאחורי בלוק זה.**
2. אם אין טיפול בחיריג זהה בבלוק הנוכחי – המשערך מփש handler בבלוק העוטף, או בקוד שקרה לשירות הנוכחי. החירג מועבר במעלה מחסנית הקראיות, אם גם ב-`main` אין טיפול, תודפס הודעה וביצוע התוכנית יסתתיים.
3. **בלוק `try` אחד יכול להיות מספר בלוקי `catch` השיבים לו**, כדי לטפל בסוגי שגיאות שונות שיכולים לקרות. **תמיד נבצע רק אחד מהבלוקים של ה-`catch`**, וזה יהיה הבלוק הראשון שמתאים לסוג השגיאה שמרתקה.

נקודות נוספות בנוגע לחזדים ולחיריגים:

1. **מחיובתו של ספק שנכשל** – שירותים מסוימים בהצלחה חייב לקיים את תנאי האחר ואת המשתרмер של המחלקה. שירות שנכשל לא חייב לקיים את תנאי האחר, **וזורק חריג כדי להתריע שתנאי אחר לא מתקיים**. **בנוסף, השירות שנכשל צריך לשחרר את המשתרмер**, מכיוון שהעוצם ממשיר להתקיים, ותכן **שירותים אחרים שלו יקראו בעמידה**.
2. **בלוק finally – יתבצע בכל מקרה**, בין אם קטע הקוד המופיע בבלוק `try` הצליח או נכשל, ובין אם קرتה שגיאה וטופלה בבלוק `catch` או שלא טופלה כלל. בבלוק זהה נוכל לשמור על המשתרмер החדש (למשל `stopFaucet()` כדי להחזיר למצח הינטראלי).
3. **יש דרך אלגנטית יותר לוודא את סגירתה המשאבים בסיום השימוש בהם, אפילו אם אין catch. לא צריך לרשום `finally`.** `try with resources` – מקרה פרטי בטיפול במשאבים – מחלוקת המממשות את הממשק `AutoCloseable`.

```

FileReader fr = new FileReader(path);
BufferedReader br = new BufferedReader(fr);
try {
    return br.readLine();
} finally {
    br.close();
    fr.close();
}

try (FileReader fr = new FileReader(path);
     BufferedReader br = new BufferedReader(fr)) {
    return br.readLine();
}

```

4. **הוכחת נכונות של ספק** – אם מוכיחים את הדיוון בחוצה לא רק לתנאי הקדם ולתנאי الآخر, אלא גם לתנאי הצד:
 - precondition & invariant & side-condition → invariant & postcondition
 - precondition & invariant & not side-condition → invariant & exception is thrown

גישהות שונות להتمודדות עם מקרים קצה

מה עושה לך שמקבל חריג?

בפונקציה `compareTo` אשר מנסה להמיר עצם מטיפוס `IPoint` לטיפוס `Comparable`, יתכן ונחטף `exception`, ספציפית מטיפוס `ClassCastException`. כדי להתמודד עם כך, נוסיף בлок של `try-catch`. אנו ניקח את השגיאה ונעטוף אותה בשגיאה אחרת, "נתרגם" את ההודעה כך שתתיה מובנת ללקוח:

```

int compareTo(Comparable other) {
    IPoint other_point;
    try {
        other_point = (IPoint)other;
    }
    catch (java.lang.ClassCastException ce) {
        throw new IncomparableException();
    }
    if (this.x() > other_point.x())

```

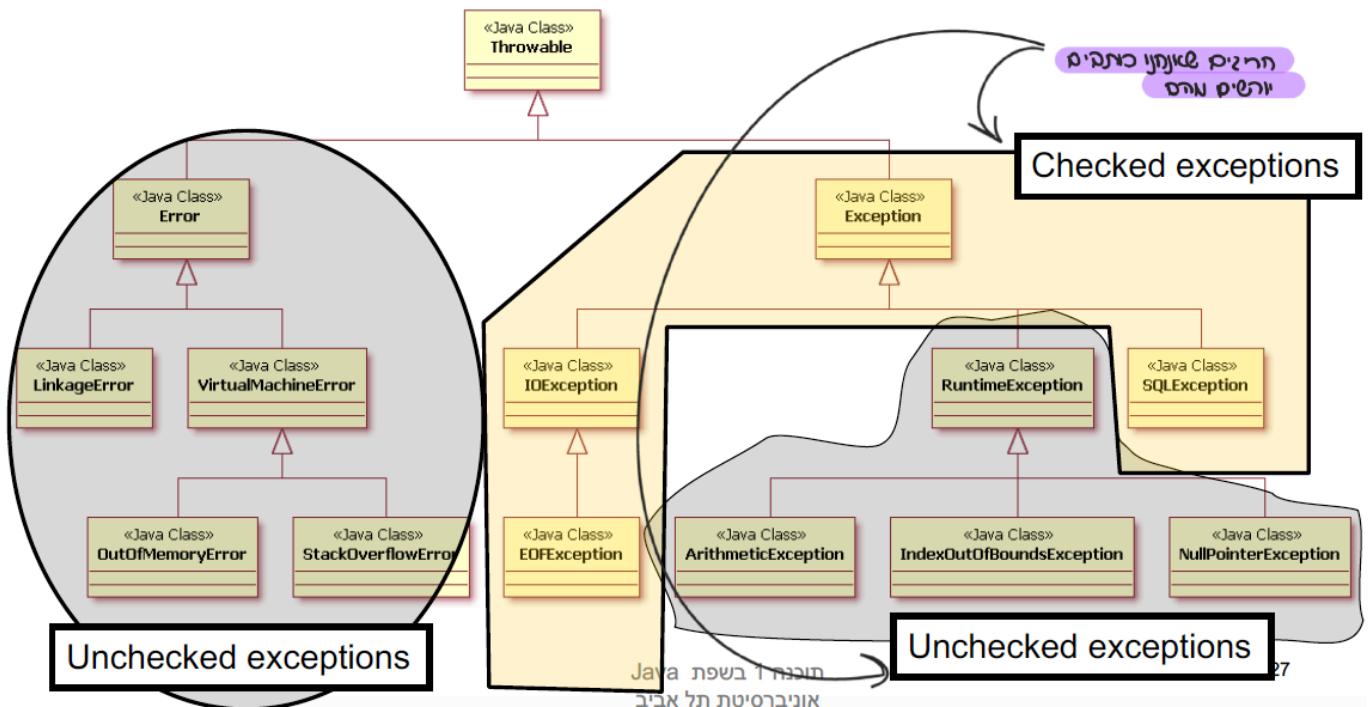
במקרה אחר, בעבודה עם קבצים, אם לא נמצא את הקובץ בשם הנtent, יוכל לנסות שם קובץ אחר:

```
FileInputStream is;
try {
    is = new FileInputStream("A:\\config.dat");
}
catch (FileNotFoundException fnfe) {
    is = new FileInputStream("A:\\config");
}
/* access the file (but only if the input stream was
   created)
*/
```

סוגי טיפוסים של חריגות:

ב-Java ההודעה על חריגת מתרחשת באמצעות עצם רגיל, שמייצג את החריג. מכיוון שהחריג הוא עצם רגיל בונום אותו בעדרת new כמו כל אובייקט אחר. הטעוג הוא לציין את הסיבה שגרמה לכישלון על ידי טיפוס כמו `java.io.FileNotFoundException`, מוגדרת הייררכיה של טיפוסים עבור חריגים, כאשר המחלקה הכללית ביותר היא `Throwable`, והחלוקה העיקרית היא:

1. Error – בעיה שלא ניתן בדרך כלל להתואושש ממנה. אולם, ניתן להגדיר חריגים מטיפוס `Error` כדי לבטא שבירה של הנחה לוגית (למשל `AssertionError`).
2. RuntimeException – חריג שיכול לקרות כמעט בכל פונקציה – גישה למצביע `null`, כישלון בהמרה, חריגה מתחום מערך וכו'. לא אמרו להופיע בתכנית תקינה.
3. Exception which isn't RuntimeException – מתרחש במצבים מוגדרים היטב, שלא ניתן למנוע אותם אבל ניתן לתכנן מראש לקרואם. למשל: `.FileNotFoundException`.



Checked exceptions: קוד המכיל קריאה למетодה שעשיה לזרוק חריג זהה, צריך לנ��ו אחת משתי הגישות: **הכרחה או טיפול** בטיפול באמצעות המילה `throws`, **try-catch-finally**, או **הכרחה או טיפול שיגיאת קומפקטציה**. השימוש במתודות הזורקות חריגים מחלחל וקורסיבית, או הכרחה או טיפול גורר שגיאת קומפקטציה.

Unchecked exceptions: על חריגים או שגיאות שהם מסווג זה, אין חובה להצהיר בחתימת המתודה. אין חובה לטפל בבלוק `try-catch-finally`. מדוע זה כך? כי הם יכולים לזרוק בכל שורת קוד, בגלל פגם בתוכנית או בגלל בעיה לא צפוייה במחשב או בסביבת התוכנה שמריצה את התוכנית. חריגים כאלה אינם צפויים ויכולם לזרוק בכל שירות. בדרך כלל הם גורמים לעצירת התוכנית ואשר זה המצב, אין חשיבות לשחזור המשתמר. אמנם, ניתן למתפוס שגיאות אלו באמצעות בלוק `try-catch`.

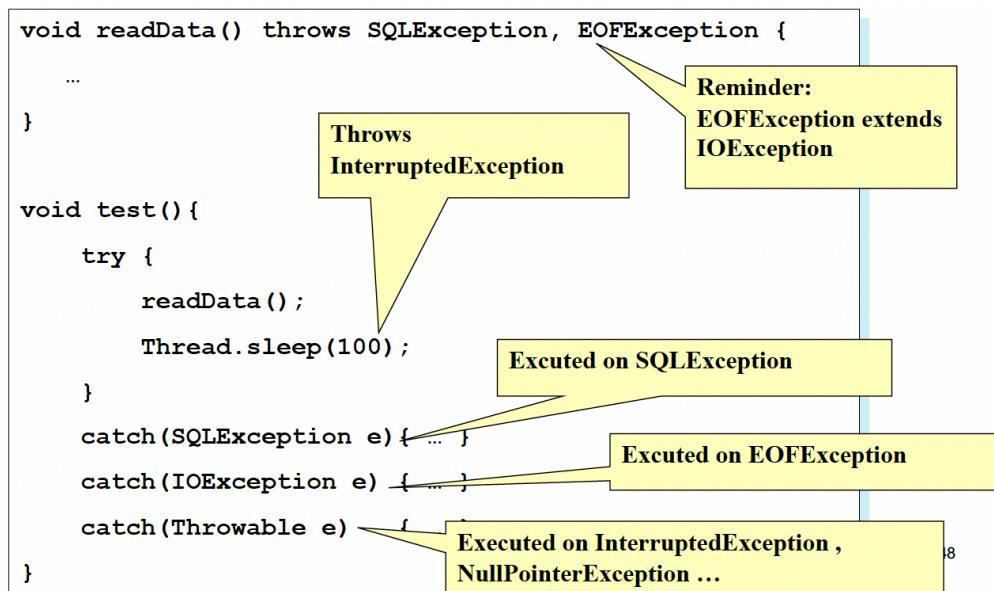
הגדרת חריגי משתמש: תחילת נחילט מאיפה אנחנו יורשים, מ-**Exception** (מחייב את כל הלקחות להצהיר או לטפל) או מ-**Error/RuntimeException** (אפשר ללקות מסוימים להעתם מהאפשרות לחיריג). חריג הוא עצם (כמו כל דבר ב-Java), והמחלקה שנגדיר עבורי יירושת מהמחלקה שבחרנו עבור סוג השגיאה. ככלים יש לפחות בניין ריק, לבני שמקבל מחרוזת, ושירות שמחזיר את המחרוזת. מקובל ליצור חריג עם מחרוזת הסבר. `getMessage`

ירושה וחריגים

נשים לב:

- למונזה דורסת/ممמשת מותר לזרוק (חריגים שיורשים) – אף חריג, חריגים שזרקה המונזה הנדרשת, חריגים היורשים מחריגים שזרקה המונזה הנדרשת.
- למונזה דורסת/ممמשת אסור לזרוק (חריגים מורשים) – חריגים שלא זרקה המונזה הנדרשת, חריגים המהווים מחלקות בסיס לחריגים שזרקה המונזה הנדרשת.

ריבוי בלאוקי catch וירושה – אם יש כמה פסוקי `catch` מתאימים יבוצע הבלוק **הראשון** **שמתאים**.



:(assertions)

- אם הביטוי `expression = false` אז התכנית תזרוק `AssertionError`. נכתב `AssertionError; assert expression;`
- טענות מבטאות הנחות שיש למתקנת על הלוגיקה הפנימית בקטע קוד מסוים – שמורה פנימית, שמורת מחלקה וכו'.

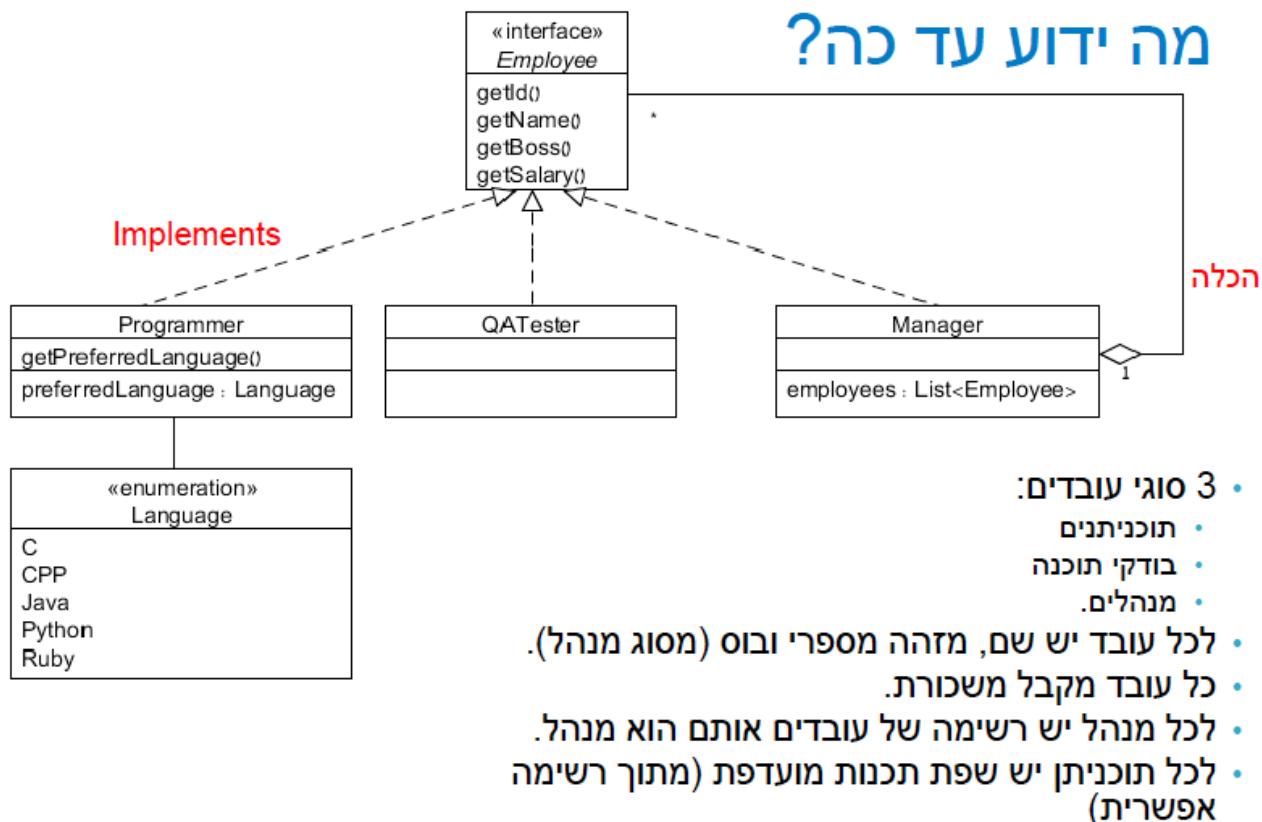
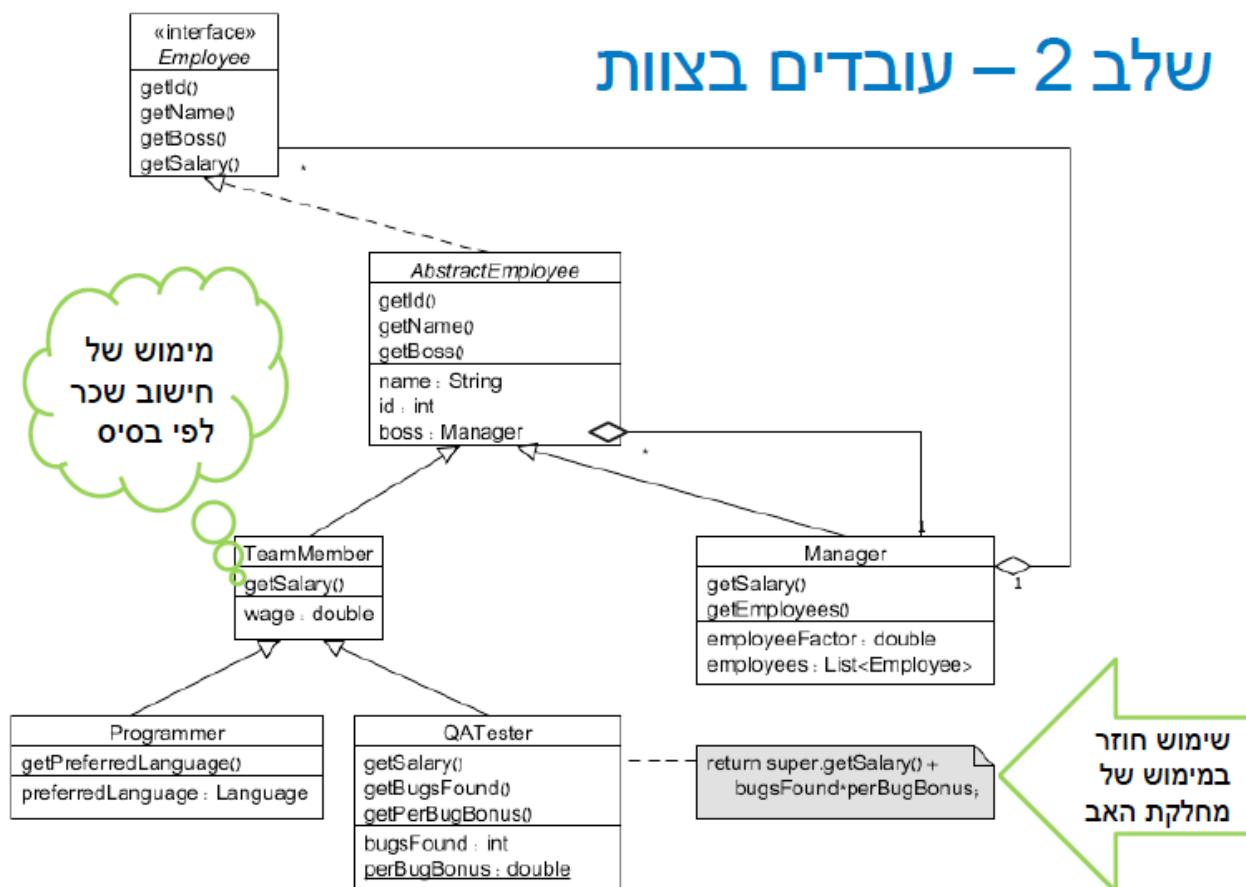
סיכום:

- חריגים מודיעים על כישלון של ספק לקיום את תנאי الآخر, למרות שהליך קיים את תנאי הקדם.
- חריג הוא מוצדק – כאשר לא ניתן לדרש מערך לקיים תנאי תנאי קדם שיבטיח את הצלחת השירות.
- חריג אינו מוצדק – אם הערך היה יכול לנמנע אותו בעזרת שאלתה פשוטה.
- טיפול בחיריג בלקוח: שחזור המשטמר והודעה ללקוח שלו על חריג (אולי אחר) או ביצוע המשימה שלו בדרך אחרת.



חברת הייטק (תרגול 9)

היררכיית המחלקות:

יצרנו מחלוקת מופשטת, אף העברנו את מימוש השבר למחלוקת נוספת `:TeamMember`



Enumerated types

```
public enum Language {
    C,
    CPP,
    Java,
    Python,
    Ruby;
}
```

וריאציה יותר מתחכמת,
הכוללת הגדרת שדות ומетодות

```
public enum Language {
    C("C"),
    CPP("C++"),
    Java("Java"),
    Python("Python"),
    Ruby("Ruby");

    private final String displayName;

    private Language(String name) {
        displayName = name;
    }

    @Override
    public String toString() {
        return displayName;
    }
}
```

METHODS important in Object

- כדי לתרום ב-`hashCode`, אנחנו צריכים לתרום **בmethodה hashCode** שתמפה אובייקט למקום מסוים. ה-`IDE` מציע השלמה אוטומטית לפונקציה זו. צריך להיעזר בשדה מסוים מהאובייקט שייהי רלוונטי. ההפרדה תהיה כאן על פי זו שהוא ייחודי.
- בנוסף נדרש לתרום **בmethodה equals**. זההות שאנו חנו קובעים כאן הוא לפי ה-`id`.

שימוש באוספים גנריים ומיפוי רשימות:

Sorting by salary

• נגידר השוואה מותאמת:

```
public class SalaryComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee o1, Employee o2) {
        return Double.compare(o2.getSalary(), o1.getSalary());
    }
}
```

מיון בסדר ההפוך – מהגדול לפחות

• כתת נול לייצר את הדוח

```
public static void printTopPaid(List<Employee> employees) {
    Collections.sort(employees, new SalaryComparator());
    for(int i=0; i<3; ++i)
        System.out.println(employees.get(i));
}
```



4 – נושאים מתקדמים

תבניות פונקציונלי וזרמיים

Enum

נרצה ליצג את כל סוגי הקלפים השונים לפי הוצאה שלהם. טיפוסים באלה, שבכל מופעיהם קבועים וידיעים מראש שכחיהם מאוד. אם ננסה ליצג טיפוס זהה באמצעות מחלקה, נגלה שדבר זה אינו בטוח ויש לו מספר חסכנות: אין שומר על בטיחות טיפוסים (אפשר לאותכל קלף עם כל ערך מספרי, ונרצה לצמצם את זה רק ל-4 ערכים אפשריים), אין שומר על מרחב שמות (אפשר לחבר בין סוג הקלף ליצוג המחרוזתי לא חלק מהמצב הפנימי), והוספת ערך חדש לטיפוס היא מורבתת.

public enum Suit { SPADES, HEARTS, CLUBS, DIAMONDS }

לכן נשתמש במבנה `enum` הפותר את בעיית הטיפוסים, **ואפשר להשתמש בו בבלוק switch-case**.

אמנם, בעית מרחב השמות עדין קיימת, וגם הוספה סוג קלף חדש מצורכה שינוי במספר מקומות. כיוון שב-`enum` כמעט כל דבר הוא עצם – ניתן להרחיב את הקונספט של `enum` להיות מעין מחלקה (עם שדות, בניאים ומethodים):

```
public enum Suit {
    SPADES("Spades"), ←———— קריאה לבנייאי
    HEARTS("Hearts"),
    CLUBS("Clubs"),
    DIAMONDS("Diamonds");

    private final String name; ←———— שדה
    private Suit(String name) { ←———— בניאי
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

כתעת אין צורך לשולוף את ייצוג
המחלקה כמחרוזת מבחווץ

הקריאה לבנייאי היא מיוחדת – אנו מגדירים את `SPADES` על ידי קריאה לבנייאי עם המחרוזת "Spades". **הבנייה הוא private, כדי שנוכל לשנות במספר המופעים שיש לנו מהאובייקט הזה. אנו לא רוצים לאפשר יצירה של מופעים חדשים מחוץ למחלקה**, אנו מייצרים את `enum` ותחממים את מספר המופעים.

התנהגות פולימורפית:

נתון לנו `enum` בשם `ArithmeticOperator` של פעולות אРИתמטיות. הפונקציה `compute` מבצעת `switch` כדי לבצע את הפעולה האРИתמטית המבוקשת:

```
public enum ArithmeticOperator {
    // The enumerated values
    ADD, SUBTRACT, MULTIPLY, DIVIDE;

    // Value-specific behavior using a switch statement
    public double compute(double x, double y) {
        switch(this) {
            case ADD:      return x + y;
            case SUBTRACT: return x - y;
            case MULTIPLY: return x * y;
            case DIVIDE:   return x / y;
            default: throw new AssertionError(this);
        }
    }
}
```

פתרון חלופי, הוא **להגדיר פונקציה מופשטת compute**, וככתבו 4 מימושים שונים של אותה הפונקציה עבור כל פעולה. `ADD` למשל הוא מופיע של-`enum`, נדרש למשב בו את הפונקציה `compute`. **פתרון זה מוצלח יותר מהפתרון הקודם**, שם נדרש לזכור שימושים נוספים פעולה חדשה, לעורך שני מקומות. בפתרון זה – **זה לא יתאפשר עד שלא נתן מימוש נוספת** ל פעולה החדשה.

EnumSet – מבנה שמכיל תתי-קבוצה של ערבי `enum` כלשהו, ומאפשר בקלות לבדוק האם הערבים אלה מתקיימים עבור אובייקט מסוים או לא. למשל עבור צורה מסוימת נאמר שהוא גם CONVEX וגם FULL. זה תופס משמעותית פחותה זיכרון, כי כל עריך מיוצג בבייט אחר.

תכונות פונקציונלי

נשים לב למחלקה האונומית הבאה:

```
Comparator<String> c = new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return Integer.compare(a.length(), b.length());
    }
}
```

מאנך מינענו את המחלקה `comparator` בלא `new`?
מאנך מינענו את המethod `compare` בלא `return`?
מאנך מינענו את המethod `Integer.compare` בלא `a.length()` ו `b.length()`?
מיסכן: מהו זה פה? מהו נגיד נגיד שמי?

מצד אחד, לא ניתן להפעיל `new` על ממשק. מצד שני, מילאנו את כל החוויות לפני הממשק, פשוט בתחריר קצר מוזר. בעת יצירת האובייקט אנחנו משתמשים את השירות `.compare`.

ממשקים פונקציונליים:

ברצוננו למיין רשימה מחזורות בסדר עולה, לפי אורכי המחרוזות. לשם כך, ניעזר **בממשקים פונקציונליים**. ממשק פונקציונלי הוא **ממשק בעל מ涕ודה מופשטת אחת בלבד** (אין אפשרות להוסיף מתודות דיפוליטיות/סטטיות), למשל כמו `Comparator`, `Iterable`. הוא אינו פונקציונלי כיון שהוא אחית – `next`, `hasNext`. הحل מ-8 Java ניתן כבר ראיינו. לעומת זאת, `Collections.sort` מחייב פונקציונלי באמצעות `lambda` (כי יש רק פונקציה אחת למשתמש).

```
Collections.sort(beatles, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return Integer.compare(a.length(), b.length());
    }
});
```

מאנך!
הגענו כרגע לכך שאין
ה-`Comparator` מושך!

אפשר לפשר את הקוד הנ"ל – נבעור לביוטי `lambda`, נמנע מיצירת המשתנה עבור ה-`Comparator` ונשלח את פונקציית ה-`lambda` ישירות ל-`sort`. אפשר גם להוריד את הטיפוסים שהקומpileר יוכל להסיק בעצמו. גם את ה-`return` אפשר להעיף ולבעור כתיבה מקוצרת:

```
Collections.sort(beatles,
    (x, y) -> Integer.compare(x.length(), y.length())
);
```

כתיבת מקוצרת עבור
מימוש בשורה אחת

لمמשקים פונקציונליים יש אנטזיה (annotation) ייעודית המביטה שהממשק מגדיר בדיק פונקציה מופשטת אחת:

```
@FunctionalInterface
public interface I1{
    public void func1(int x);
}
```

רפרנסים למתודות:**■ רפרנס למתודה סטטית:**

```
List<Integer> ints = Arrays.asList(5,1,2,4,3);
ints.sort(Integer::compare);
```



```
ints.sort((x,y)->Integer.compare(x, y));
```

■ רפרנס למתודה מופעת:

```
List<String> strings = Arrays.asList("aa", "Ab", "BA", "Bb");
strings.sort(String::compareToIgnoreCase);
```

גיאן גאנט פונקציית קומפקטיות
בזווית זו קומפקט



```
strings.sort((x,y)->x.compareToIgnoreCase(y));
```

דוגמה:

במחלקה Tree, במימוש הֆונקציות sumLeftValues, sumRightValues אנו נתקלים בדימויים רבים שבהם רבו מאוד עד כדי שורת קוד אחת. מדובר בשכפול קוד גדול פרט לקריאה לפונקציה של Node (שמאליה או ימינה). אפשר להעביר ארגומנט בוילאי של isLeft או באופן כללי זה עדין לא מספיק טוב. יותר נכון להעביר את הֆונקציה עצמה, שתחלץ את **ה-node** הבא בכיוון הרלוונטי. במקרה זה ב-Java נוכל להשתמש במנשך פונקציונלי!

זרמים

זום הוא סדרה מופשטת של אלמנטים התומכים ביצוע פעולות צבירה (aggregation), באופן סדרתי (יש תלות באיבר, עושים פעולה על כל איבר בנפרד) או מקבילי (ביצוע פעולה על כל האיברים ביחד). בקורס זה נדבר על **ביצוע פעולות סדרתי**.

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
ints.stream().map(x->x*x)
    .filter(x->(x%2 == 0))
    .forEach(System.out::println);
```



```
.forEach(x->System.out.println(x));
```

<u>output:</u>
4
16

*תען כי לא ניתן לגשת
הזרם לא נסעה.
*מפטון - קונה נאוץ
ללאו קונה לא נאוץ
ללאו קונה כנו זום

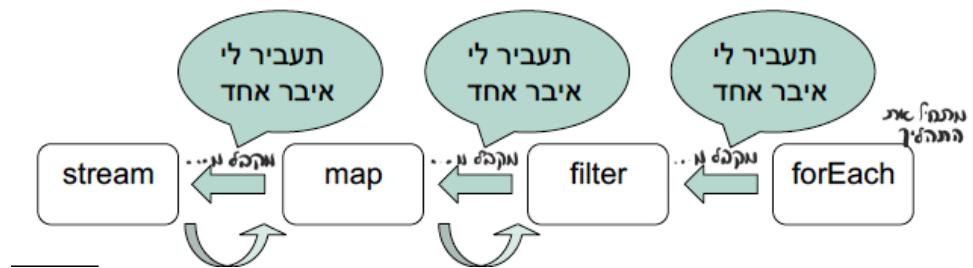
מה יודפס בהרצה
הקוד? גאיםו וטעו?

ניתן לחלק את הפעולות על זרמים לשתי קבוצות:

1. **פעולות ביניים (intermediate)** – פעולות אלה מופעלות על זרמים ומחזירות זרם, כך שניתן לשרשר אותן אחת לשניה. למשל בדוגמהelow הֆונקציות map, filter.
2. **פעולות סופניות (terminal)** – פעולות אלה יופיעו בסוף שרשרת פעולות על זרם, כמו למשל ניתן לשרשר אחריהן פעולות מסוימות על אותו זרם. בדוגמהelow מדבר בפועלה forEach. בזמן שלא צורכים איבר של הזרם – לא קורה כלום. מי שוצרך איברים של הזרם הם שירותים סופניים. **הפעלת שירות סופני גורמת לזרם לייצר את האיברים שלו.**



אם נוריד את הקראיה ל-`forEach` לא יתחל התהליך כל, הוא זה שיוצר את כל השרשרת:



דוגמה נוספת (אנו רואים כיצד הפעולות מופעלות איבר-איבר בזרם):

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
ints.stream().map(x->{
    System.out.println("mapping " + x);
    return x*x;
})
.filter(x->{
    System.out.println("filtering: " + x);
    return x>10;
})
.forEach(x->{
    System.out.println("terminating: " + x);
});
```

output:
mapping 1
filtering: 1
mapping 2
filtering: 4
mapping 3
filtering: 9
mapping 4
filtering: 16
terminating: 16
mapping 5
filtering: 25
terminating: 25

מה יודפס?

פעולות נוספות על זרים:

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
boolean res = ints.stream()
    .map(x->""+x)
    .skip(2)
    .anyMatch(x-> x.equals("1"));
```

שינוי טיפוס האיברים בזרם מ Integer ל String
דילוג על שני האיברים הראשונים בזרם
אם לפחות אחת המחרוזות בזרם היא המחרוזת "1"?

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
boolean res = ints.stream()
    .limit(1)
    .allMatch(x-> x ==1);
```

מגבילים את מספר האיברים בזרם ל 1
אם כל האיברים בזרם שווים ל 1?

.`forEach`, `anyMatch`, `allMatch`, `noneMatch` הן פעולות סופניות, כמו `each`.
בניגוד ל `forEach`, הן מחזירות ערך בוליאני.

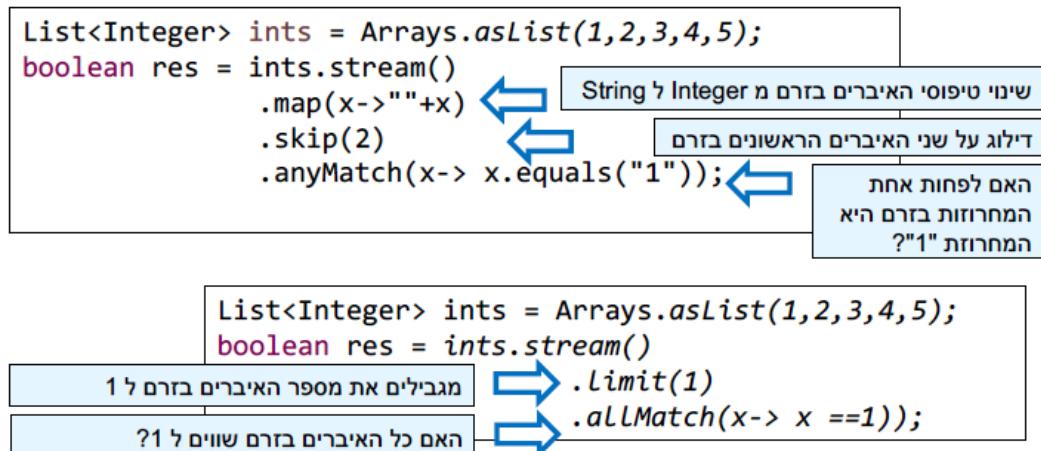


פעולות נוספות על זרים:

1. פעולות סופניות – `.noneMatch, allMatch, anyMatch`
2. פעולות נוספת – `.skip, limit, peek, sorted`

כאשר מפעילים פונקציות על זרים באמצעות פקודות כמו `map` וכו', נדרש להיות להן שתי תכונות:

1. Non-interfering – מאפשר לשנות את האוסף עליו מופעל חומר (הוסף/חוות אובייקטים).
2. Stateless – חסנות תופעות לוואי. תוצאות הפעולות צריכה להיות תלויה באיבר הזרם עליו היא מופעלת, ולא במצב של שדה או משתנה אחר.



נשים לב ש-`sorted` מחייב שכל האיברים יעמדו עליו מה-`peek` כי הוא יכול לפעול רק על כל הזרם, ואז הוא ימיין אותם. רק לאחר מכן פועלות המיוון הסטימיט, `forEach` מופיעות אולם `sorted` היא מיוחדת בהקשר זהה ולא מקבלת איבר-איבר, אלא מקבלת את כל הזרם, ואז מחדירה את כלו ממויין.

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

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

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

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

מה ישנה אם נבצע את פעולה `filter` ?sort ?



נקודות נוספות:

1. `<T>Optional<T>` – פתרון אחר למצב שבו היינו אול רצים לזרוק exception. מדובר בפונקציה שלפעמים רוצה להחזיר תשובה, אבל לא תמיד: למשל חיפוש איבר במערך והחזרת האינדקס שלו, צריך להתייחס למקרה הקרה שבו לא מצאנו את האיבר. השימוש במחלקה הגדנית `Optional` מאפשר לנו לבטא באופן אחד מצב שבו הפונקציה לא מחזירה ערך. אם נגידו שהפונקציה מחזירה `Optional<T>` עבור טיפוס גנרי `T` כלשהו, אנחנו נא מתחייבים להחזיר ערך מהטיפוס הזה, אויל נחזיר ערך "ריק/null" שמצויד באמצעות `Optional.empty()`. בהתאם, נחזיר את הערך המקורי והרצוי גם בעטיפה של המחלקה `Optional<U>` `Optional.of(result)`.

```
public static Optional<String> findStringOfLenK(
    List<String> strings, int k){
    for (String s: strings){
        if (s.length() == k){
            return Optional.of(s);
        }
    }
    return Optional.empty();
}
```

שימושים מעוניינים ומקרים בערך האופציונלי:

```
List<String> lst = Arrays.asList("John", "Paul", "George", "Ringo");
Optional<String> strOfLen6 = findStringOfLenK(lst, 6);
if (strOfLen6.isPresent()){
    System.out.println(strOfLen6.get());
}

strOfLen6.ifPresent(System.out::println);

Optional<String> strOfLen3 = findStringOfLenK(lst, 3);
System.out.println(strOfLen3.orElse("no-value"));
[לעומת צהוב]
```

output:
George
George
no-value

2. פעולת reduce – זהה פעולה סופנית, המחזירה ערך אופציונלי (טיפוס `Optional`). אם הזרם ריק, היא אינה מחזירה ערך (ערך ריק). אחרת, היא מחזירה תוצאה של **ציבורת כל האיברים בזרם** (מצומם שליהם לאיבר אחד) באמצעות הפונקציה אותה היא מקבלת כפרמטר (חיבור כל האיברים, נפל כל האיברים, **לקיחת האיבר המקסימלי/המינימלי**, או כל פעולה אחרת שהפונקציה גורמת לה לבצע). אם הזרם מכיל איבר יחיד, פועלות ה-`reduce` ו-`reduce` את האיבר זהה.

```
List<Integer> ints = Arrays.asList(1,2,3,4,5);
Optional<Integer> product = ints.stream()
    .reduce((x,y)->x*y);
Optional<Integer> sumOfSquares = ints.stream()
    .map(x->x*x)
    .reduce((x,y)->x+y);
```

3. טיפוסים גנריים בירשה/מיימוש – הממשק `BinaryOperator` הוא מנשך פונקציונלי. את הפונקציה המופשטת שלו `apply` הוא יורש מהמנשך `BiFunction`. בעצם, מה **شمגדיר BinaryOperator** הוא משתנה גנרי יחיד, וכל הטיפוסים דומים בפונקציה שלו – **שני הארגומנטים, והתוצאה**.

```
@FunctionalInterface
public interface BinaryOperator<T>
    extends BiFunction<T,T,T>
```

```
@FunctionalInterface
public interface BiFunction<T,U,R>
```

Represents a function that accepts two arguments and produces a result.

This is a functional interface whose functional method is `apply(Object, Object)`.

המנשך `BiFunction` מגדר שלושה פרמטרים גנריים (`T,U,R`), בעוד שהמנשך `BinaryOperator` מגדר פרמטר גנרי אחד בלבד. לכן, `BinaryOperator` נדרש לבצע הצבה לתוך הפרמטרים `T,U,R` של `BiFunction`. הוא מציב בשולשתם את ערכו של `T`, הפרמטר הגנרי שלו. המשמעות: הפונק' `apply` של `BinaryOperator` מופעלת על שני איברים מאותו הטיפוס ומחזירה את אותו הטיפוס.



4. יצירת זרם אינטפי – המנשך `<T> Supplier` מתאר זרים של איברים מטיבוס `T`. זרם מוגדר על ידי פונקציה אחת שהיא `anyMatch`. על דברים אינטפיים קשה לו להחזיר `false`, ומשיר לroz. אם הוא מצא 1 הוא ישר לחזיר `true`.
שימוש בזרם האינטפי:

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
s.limit(5).reduce((x,y)->Math.max(x,y))
```

ifPresent
reduce Optional

output:
5

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
s.map(x->x+1).forEach(System.out::println);
```

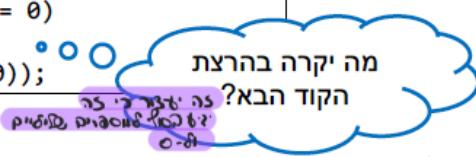
■ מבין המספרים המתחלקים ב 7, האם קיים מספר המתחלק ב 10 ?

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
System.out.println(s.filter(x-> x % 7 == 0)
    .anyMatch(x-> x % 10==0));
```

output:
true

■ מבין המספרים המתחלקים ב 7 והקטנים מ 70, האם קיים מספר
המתחלק ב 10 ?

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
System.out.println(s.filter(x-> x % 7 == 0)
    .filter(x-> x < 70)
    .anyMatch(x-> x % 10==0));
```



5. איסוף – הfonקציה `collect` מייצרת אוסף מהזרם, אפשר להשתמש ב-`Collectors.toList()` (המרה לרשימה), או למשל `Collectors.averagingDouble` (חישוב ממוצע).

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
Double d = s.limit(5).collect(Collectors.averagingDouble(x->x*x));
System.out.println(d);
```

מפעילה את `averageingDouble`
הfonקציה שמקבל כפרמטר על אברי הזרם
ומчисבת את הממוצע שלהם

Output:
11.0



איסוף מתקדם יותר הוא `partitioningBy` (מציב תנאי לחולקה הזרם לשתי קבוצות לפי הפסיקט, המילון ימפה את ערך התוצאה של התנאי (true/false) לרשימה של האיברים בהתאם לתוצאה זו).

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
Map<Boolean, List<Integer>> partition = s.limit(5)
    .map(x->x*x)
    .collect(Collectors.partitioningBy(x->x>10));
System.out.println(partition);
```

output:
`{false=[1, 4, 9], true=[16, 25]}`

זה מקרה פרטי של `groupingBy` (באנו מחלקים ליותר משתי קבוצות, ובמיון תמורה כל תוצאה לרשימה הערכים שמקיימים אותה).

```
List<String> beatles = Arrays.asList("John", "Paul", "George", "Ringo");
Map<Integer, List<String>> groups = beatles.stream()
    .collect(Collectors.groupingBy(x->x.length()));
System.out.println(groups);
```

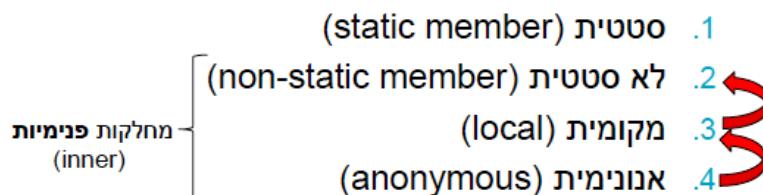
output:
`{4=[John, Paul], 5=[Ringo], 6=[George]}`

מחלקות מקוננות -Binding (תרגול 10)

מחלקות מקוננות:

- מחלקה מקוננת סטטית היא אינה מחלקה פנימית (לא מושרhta לשום מופע של המחלקה העוטפת). אם לא נתן לה מופע של המחלקה העוטפת, היא לא יכולה לגשת לשודות מופע שלה, היא יכולה לגשת רק לשודות static.
- דוגמה קלאסית למחלקה לא סטטית: חדר, הקים בתוך המחלקה העוטפת בית. אין מופע של חדר ללא מופע של בית!
- מחלקה מקוננת היא מחלקה המוגדרת בתחום מחלקה אחרת.

סוגים:



מחלקה סטטית – לא קשורה ל-instance של המחלקה העוטפת.
 לא סטטיות – כן קשורות ל-instance של המחלקה העוטפת.

- כדי לגשת לשדה של House מתוך Room ניתן לבתוב בכתיב מלא :House.this

```
public class House {
    private String address;
    private double height;

    public class Room {
        private double height;
        // implicit reference to a House
        public String toString() {
            return "Room height: " + height
                + " House height: " + House.this.height;
        }
    }
}
```

Shadowing

Height of Room
Same as this.height

Height of House



- אפשר ליצור "שירות" (בשורה אחת) מופע שלמחלקה פנימית (לא סטטית) אחרי יצירת מופע של המחלקה החיצונית:

```

public class Parent {

    public static class Nested{
        public Nested() {
            System.out.println("Nested constructed");
        }
    }
    public class Inner{
        public Inner() {
            System.out.println("Inner constructed");
        }
    }
    public static void main(String[] args) {
        Nested nested = new Nested();
        Inner inner = new Parent().new Inner();
    }
}

```

Construct nested
static class

Construct nested
class

חסוב מאוד ל מבחני!

:static binding

- מתודות static – בהתאם לטיפוס הסטטי של המשתנה, זה שמוגדר בצד שמאל – .ABAA

```

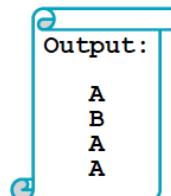
public class A {
    public static void m() {
        System.out.println("A");
    }
}

public class B extends A {
    public static void m() {
        System.out.println("B");
    }
}

public class StaticBindingTest {
    public static void main(String args[]) {
        A.m();
        B.m();

        A a = new A();
        A b = new B();
        a.m();
        b.m();
    }
}

```



- שודות – גם בהתאם לטיפוס הסטטי של המשתנה! נלק לפיו מה שמוגדר בצד שמאל.

```

public class A {
    public String someString = "member of A";
}

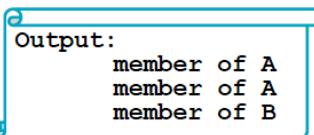
public class B extends A {
    public String someString = "member of B";
}

public class StaticBindingTest {
    public static void main(String args[]) {

        A a = new A();
        A b = new B();
        B c = new B();

        System.out.println(a.someString);
        System.out.println(b.someString);
        System.out.println(c.someString);
    }
}

```



- גם static binding שאי אפשר לדרכו. טיפ: לסמן S ליד השורה כי מדובר ב-

:dynamic binding

- רק בזמן ריצה נדע באיזה אובייקט אנחנו נמצא! או אפשר לדעת מראש לפי הטיפוס הסטטי.
- שאלה מבחינה:

שאלה מבחינה:
 .D – goo ○
 .D – func ○
 .S – foo ○
 היא פונקציה של !Base
 moo – D. ○
 moo – Sub. ○
 נדרשה והולכים ל-Sub.

שאלה מבחינה (2021 ב', מועד א')

```

public class Base {
    public int func(){ return 2 + foo(); }      2. no func in Sub
    private int foo(){ return 5 + moo(); }        3. foo is private
    public int moo(){ return 1; }
}

public class Sub extends Base {
    public int goo(){ return func() + foo(); }   1. no goo in Base
    private int foo(){ return 3; }                 5. foo is private
    public int moo(){ return 2; }                  4. moo is not
    public static void main(String args[]){       private/static/final
        Sub sub = new Sub();
        System.out.println(sub.goo());
    }
}
  
```

2+5+2+3=12

מה יודפס בהרצה התכנית Sub ?



ירושה III (מתקדמת)

טעינה לפי טיפוס דינמי/סטטי

דוגמה 1:

המחלקה Cat יורשת מ-Animal שני מетодים: אחת סטטית - hide, ואחת של מופע - override. היא דורשת את שתי הפונקציות.

```
public class Client{
    public static void main(String[] args) {
        Cat myCat = new Cat();
        Animal myAnimal = myCat;
        //myAnimal.hide();      //BAD STYLE
        Animal.hide();         //Better!
        myAnimal.override();
    }
}
```

מה יודפס?

**The hide method in Animal.
The override method in Cat.**

בשפת Java
וגם בשפת תיל אבב

- עברו () – פונקציה סטטית תיקרא לפי הטיפוס הסטטי. בברזון קומpileציה אנו יודעים איזה קוד יירוץ, כי אין צורך בטיפוס הדינמי.
- עברו () – פונקציית מופע תיקרא לפי הטיפוס הדינמי.

דוגמה 2:

נתבונן במחלקות Sub, Base. ב-Sub אנו דורסים את `priv`, אבל אנחנו לא דורסים את `priv` (כי היא private ב-Base, ואו אפשר לדרש פונקציות שאן `priv`, הזרות של הפונקציה נקבעת בזמן קומPILEציה).

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

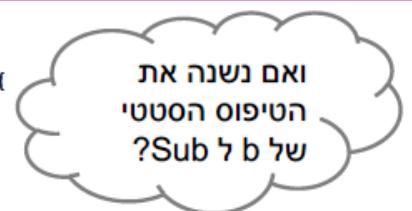
    public void foo() {
        priv();
        pub();
    }
}

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

כאשר נבצע `b = new Sub()` ונשים לב כי יקרה `priv` של Base, ולאחר מכן `pub` של Sub (לפי הטיפוס הדינמי, כי פונקציה זו אכן נדרשה).

```
public class Test {

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



במקרה שבו נשנה את הטיפוס הסטטי ל-Sub, עדין תיקרא הפונקציה `priv` של Base – כיוון שבעתה הקראיה ל-`priv` אנו בעצם מבצעים `(this).priv()` והטיפוס הסטטי של `this` הוא Base, כי זו המחלקה שבה כתוב הקוד (על אף שיש גם את `b` שמצויב לאותו האובייקט אבל רואה אותו מנוקדת מבט אחרת, של Sub). `priv` אינו נדרס (כמו שאמרנו קודם) ולכן לא יקרה `priv` של Sub.

דוגמה 3 – שדות, הורשה ו קישור סטטי:

גם קומpileציה של התיאחשות לשדות מתבצעת בצורה סטטית. מחקה יורשת יכולה להגדיר שדה גם שדה בשם זה היה קיים במחלקה הבסיס (מאותו טיפוס, או מטיפוס אחר). תמיד נקבע לאיזה שדה אנו הולכים לפי הטיפוס הסטטי! אין דבר כזה דריש שדות, הם תמיד מתואספים אחד לשני.

אם נרצה לגשת לו של `bs` בתור `Sub`, נצטרך לבצע המרה לפני מהה: `i`.`((Sub)bs).i`.

```

public class Base {
    public int i = 5;
}

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

5
five
5

```

```

public class Test {

    public static void main(String[] args) {
        Base bb = new Base();
        Sub ss = new Sub();
        Base bs = new Sub();

        System.out.println(bb.i);
        System.out.println(ss.i);
        System.out.println(bs.i);
    }
}

```

מה יודפס?

הסקת טיפוסים במשתנים מקומיים

במקרים מסוימים, ניתן לוותר על הצהרת הטיפוס הסטטי של משתנה מקומי. החל מ-10 Java נוכן **להשתמש ב-var** על מנת לkür את הכתיבה ולחסוך טיפוס משתנה ארוך:

- רק עבור משתנים **שקיים עבורם אתחול** בגין ההצהרה על המשתנה. אין כאן שום דבר שמסביר מה הטיפוס הסטטי.

var i;

משתנה זה צריך להיות מאותחל כך **שייה ניתן להסיק מהאתחול את הטיפוס**. בדוגמה השנייה לא ברור איזה מנשך פונקציוני ממשים כאן, אין דרך לדעת.

var i = null;
var j = (String x -> x.length());

- לא ניתן להגדיר פרמטר לפונקציה מטיפוס `var` – הפונקציה לא מדע לעבוד איתם.
- הטיפוס נקבע לפי האתחול של המשתנה.

var lst = new PolarPoint(...); //lst is of type PolarPoint

- לא ניתן לשימוש עם טיפוס גנרי.
- לא ניתן להשתמש ב-`var` כהצבה לפרמטר גנרי.

List<var> i = new ArrayList<String>();

- כן ניתן להשתמש ב-`var` בשביל טיפוסים גנריים אבל בזיהירות! במקרה השני הטיפוס יהיה `.Object`.

var strArrList = new ArrayList<String>();
//strArrList is of type ArrayList<String>
var objArrList = new ArrayList<>();
//objArrList is of type ArrayList<Object>



נדע על Generics

איך זה עובד?

באשר יש לנו מחלקה גנרטיבית `<Something>`, הקומpileר מפנה אותה למחלקה אחת וריגלה (לא גנרטיבית) שהיא בעצם `<FCStack<Object>`. בקוד שמשתמש במחלקה מובילה, הקומpileר מוסיף לקוד המרות, על מנת לבצע השמות מ-Object הבלתי לטיפוס ספציפי כלשהו למשל `String`. הקומpileר מוסיף **שההמרה תמיד תצליח (לטיפוס הספציפי, לא לכל דבר)** ולעולם לא תodium על **ClassCastException**. בדוגמא, הטיפוס המובליל (`T`) נמחק מהקוד שהקומpileר מייצר, והוא שימושי רק לבדיקות תקיןות טיפוסים בזמן קומpileציה. התהילך נקרא **מחיקה (erasure)**.

בטיחות טיפוסים:

```
Stack <String> ss = new FCStack <String> (5);
 ss.push("The letter A");
 ss.push(new Integer(3));
 String t = ss.top(); // same as: (String)ss.top();
```

■ מכיוון שרק מחרוזות יכולות להיות מוכילות במחסנית אין עוד צורך בהמרה

```
Stack <Rectangle> sr = new FCStack <Rectangle>(5);
Rectangle rr = new Rectangle(...);
Rectangle rc = new ColoredRectangle(...);
ColoredRectangle cc = new ColoredRectangle(...);

 sr.push(rr);
 sr.push(rc);
 sr.push(cc);
```

: הכללה ויחס a-is-a

```
Stack <String> ts = new FCStack <String> (5);
Stack <Object> to = new FCStack <Object> (5);
 to = ts;
 ts.push("The letter A");
 ts.push(new Integer(3));
 to.push(new Integer(3));
```

■ מסקנה: `FCStack<Object>` **אינו** סוג של `FCStack<String>` ■
 ■ זה לא אינטואיטיבי אבל נכון.

- בין מחסנית של מחרוזות למחסנית של אובייקטים לא מתקייםיחס a-is-a. שימוש שגוי בהמרה מהסוג הזה יגרמו לשגיאת זמן ריצה. נראה היגויו של אובייקטים לצבע למחסנית של מחרוזות, אבל יש דברים שאפשר לבצע על ts או ai אפשר לבצע על to. **אנו** אפשר לדוחוף ל-ts מספר שלם, אבל ל-to כן אפשר. אם היינו יכולים לבצע את ההצבעה זו, יוכל לשים במחסנית של מחרוזות מספר שלם!
- אם היה מדובר פשוט ב-Object ו-String לא תהיה בעיה לבצע המרה כלפי מעלה. הבעיה היא כיון שהוא מדובר במבנה נתונים שעשו שימוש בטיפוסים האלה, והפונקציה (s)push(String) אוסף נמיר כלפי מעלה את מחסנית המחרוזות למחסנית של Object, יוכל לקרוא ל-spush עם מהו שיופיע מ-object: נקבל את הפונקציה (s)push(Object) ואז יוכל לספק לפונקציה זו עם מחלוקת שלא בהכרח יורשת מ-String!Integer כmo **!String**
- באופן כללי – עדיף לא לערבב מערכיים עם טיפוסים גנרטיבים: מערך עם טיפוס גנרי T לא ניתן ליצור, בגין מחיקת הטיפוסים בזמן ריצה. **מומלץ להימנע משימוש במערכות גנרטיבים**, ולעבוד עם אוסףים גנרטיבים במקום זאת.

סוגיות נוספות:

1. **טיפוסים נאים (raw types)** – מנגנון הגנריות נוסף מאוחר, ולכן היה צריך לאפשר שימוש במחלקות גנריות גם מקוד ישן שAIN ב הכללות שבאלו. לכן אפשר לתמוך בכך בעבר באמצעות שימוש במחלקות ישירות ללא פרמטר. לא מדובר בפיצ'ר מבחרינו שצורך להשתמש בו, רק תופעה שכיריה להכיר. כל השורות באן מתקפלות. **ניסיונות לבשה raw** זהה שקול ל-Stack<Object>, והוא זורם בקומpileציה עם הכלול, אולם בזמן ריצה נפגוש את השגיאות.

```
class FCStack <T> implements Stack <T> { ... }

...
Stack <String> vs = new FCStack <String>();

Stack raw = new FCStack(); //What about T?

raw = vs; // ok
vs = raw; //uncHECKED compiler warning
```

2. **גבול עליון (extends)** – גבול עליון הוא שם של מחלוקת או מחלוקת שמננה ירוש הטיפוס הגנרי. למשל, באשר הגבול העליון הוא Object, לא ניתן לבצע פעולה על עצמים מהטיפוס הגנרי. על כן, בהגדרת טיפוס גנרי ניתן לספק גבול עליון אחר. לעומת זאת, ה-T צריך להיות מטיפוס של דבר כלשהו שיורש מ- Comparable (מתקיים יחס-a-is).

```
public class SortedSetImplementation<T extends Comparable> {
    ...
    T elem1 = ...
    T elem2 = ...
    elem1.compareTo( elem2 ) ....
    expectComparable(elem1); //elem1 is indeed Comparable
}
```

בכך ששםנו גבול עליון ל-T, אנו יודעים שנייתן לקרוא בעת לפונקציה compareTo.

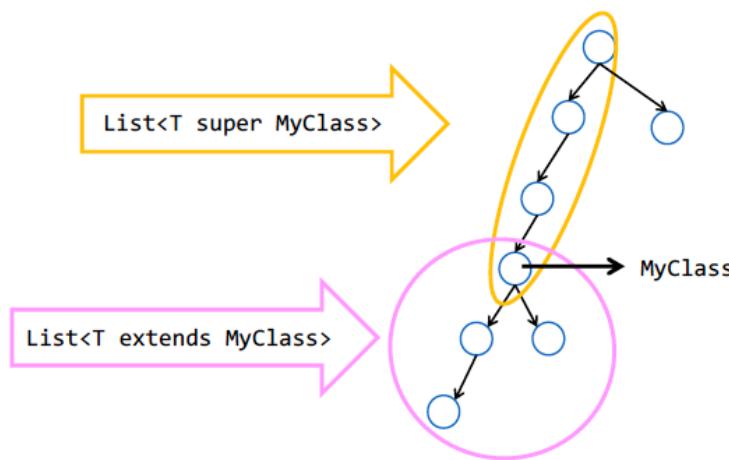
3. **Comparable גנרי – שימוש ב-extends Comparable** – שימוש שהוא raw זה בעייתי. תכנו שני עצמים שככל אחד מהם Comparable אבל הם אינם Comparable אחד עם השני. לנעדי גרסה גנרית וניצין בmphorsh: MyClass implements Comparable<MyClass>. בוצרה זאת מגדירים מחלוקת שעוצמה ברוי השוואת עצם עם עצמו. ומספקים שירות שմבצע את ההשוואה.
4. **ג'וקרים (?)** – אם נרצה לשלב קוד ישן שמכיל List בקוד חדש, ננסה לשדרג את הטיפוס לטיפוס מוככל List<Object>. אמנם זו בעיה, כי לא נוכל להעביר למשל List<String>. لكن נשתמש בסימן שאלה, וכך הפונקציה מקבל כל טיפוס של List. כדי שנוכל לבצע פעולות על איברי הרשימה, יש לספק חסם עליון (באטען extends Comparable) בפי שראים).

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

■ **משמעות ההגדירה:** הטיפוס הגנרי של list הוא טיפוס המרחיב את IShape כולל IShape עצמו כמובן.

■ **שימוש לבן לשימוש ב-extends גם עבור מנשיים.** זהו תחביר מיוחד להרחבות.

באופן דומה ניתן גם לספק חסם תחתון (באטען super).



5. **שירותים מוכליים** – ניתן להגדיר פרמטר גנרי רק עבור פונקציה מסוימת, ולא רק למחלקה. החתימה של הפונקציה נראית כך: `(public static <T> T getFirstItem(List<T> list)`. באשר בקשר לפונקציה זו, היא מסיק את הטיפוס של `T` לפי הרשימה שנשלחה אליה. ה-`T` הראשון `<T>` הוא פשוט סימן לכך שמדובר במתודה גנרית, ו-`T` לאחר מכן הוא טיפוס ערך החזרה של הפונקציה.

■ האם פרמטר גנרי של פונקציה זהה לג'וקר?

■ במקרים מסוימים תיתכן התנגדות זהה, אבל כלל, שימוש בפרמטר גנרי מאפשר קיבוע של הטיפוס הגנרי ושימוש בו ביותר מקום אחד.

```

 public <T> void f1(List<T> list) {
    list.add(list.get(0));
}

 public void f2(List<?> list) {
    list.add(list.get(0));
}

```

`T elem = list.get(0);`
`list.add(elem);`

`Object elem = list.get(0);`
`list.add(elem);`

לא נוכל להוסיף `Object` לרשימה של `String` למשל:

■ האם השירותים הבאים זהים?

```

Public static <T> void f1(List<T> l1, List<T> l2) { }

Public static void f2(List<?> l1, List<?> l2) {}

```

```

public static void main(String[] args) {
    List<String> l1 = null;
    List<Integer> l2 = null;
     f1(l1, l2);
     f2(l1, l2);
}

```



6. **מחלקות פנימיות** – נשים לב להבדל בין מחלוקת פנימית סטטית ולא סטטית. בנוסף, נשים לב כי ניתן להשתמש ב-<?> גם לא בפונקציה, גם בהגדלה של משתנה! זה אותו דבר בדיק. זה תחביר חוקי בשפה, וזה אומר שהטיפוס יכול להיות כל דבר.

```
public class MyType<E>{

    class Inner{}
    static class Nested{};

    public static void main(String[] args) {
        MyType mt; //warning: MyType is a raw type
        MyType.Inner inn; //warning: MyType.Inner is a raw type
        MyType.Nested nest; //no warning, not a parametrized type
        MyType<Object> mt1; //no warning
        MyType<?> mt2; //no warning, ? is OK for a type
    }
}
```

Raw Showna מ-<Object> וגם מ-<?>. אם משתמשים בו, הכל מתקמפל ואז עדיף לצפות שגיאות שיכולה לקרות (נגלה אותן רק בזמן ריצה). לכן עדיף לא להשתמש ב-Raw. מנגנון של טיפוסים גנריים בודק יותר ועלה שגיאות קומpileציה אם ישן בכלל.

```
public static void main(String[] args){
    List<String> strLst = new ArrayList<>();
    appendNewObject(strLst); //compilation error!
}

public static void appendNewObject(List<Object> lst){
    lst.add(new Object());
}
```

מה היה קורה אם הפונקציה appendNewObject הייתה מקבלת List נא?

מנגןן מחיקת הטיפוסים:

נוקח מחלוקת גנרית <T>. בזמן קומpileציה הטיפוס הגנרי T מוחלף בגבול העליון. הגבול העליון הוא Object אלא אם צוין אחרת בהגדרת הטיפוס הגנרי. לכן בעצם T הופך ל-Object. **בכל מקום שבו היה T הופך להיות Object המחלוקת כבר לא גנרית, והטיפוס נעלם, היא הופכת לraw.** אבל איך בא לידי ביטוי העניין של String (טיפוס שהעבכנו למחלוקת)? מתבצעת המרה מ-Object getT() שמחזירה String לטיפוס Object (마חריו הקלים).

```
public static void main(String[] args){
    Gen<String> b = new Gen<>("abc");
    String item = b.getT();
}
```



```
public static void main(String[] args){
    Gen b = new Gen("abc");
    String item = (String)b.getT();
}
```

הקוד שעושה שימוש במחלוקת הגנרית לפני ואחרי הקומpileציה



סוגיה נוספת, היא התייחסות מיוחדת בעט ירושה מ-`Gen<T>`, והגדרת מחלקה `SGen extends Gen<String>`. לאחר קומpileציה, הפעמיון הגנרי נעלם בשתי המחלקות. מה הבעה? במחלקה `Gen` הפונקציה `setT(Object t)` מקבלת אובייקט `Object` לאחר מכן הפעמיון הגנרי, ועוד במחלקה הירושת `SGen` הפונקציה `setT(String t)` מקבלת אובייקט `String` (!). הקומpileר מייצר פונקציית גשר.

הקומPILEר מייצר פונקציית גשר:

```

public class Gen {
    Object t;
    Gen gen;

    public Gen(Object t) {
        this.t = t;
    }

    public Object getT() {
        return t;
    }

    public void setT(Object t) {
        this.t = t;
    }
}

public class SGen extends Gen{
    public SGen(String t) {
        super(t);
    }

    public void setT(Object t) {
        setT((String)t);
    }

    public void setT(String t) {
        System.out.println(t);
        super.setT(t);
    }
}

```

(תרגול 11) Generics

תכונות גנריות:

- יתרונות על פני שימוש ב-`Object` – לכל מופע נרצה לוודא שככל מקום בו הופיע `T`, יהיה שימוש באותו הטיפוס `T` בדיק. אחרת,
- אפשר למשל להבניס לאותה רשימה גם מחרוזות וגם מספרים.
- גם מנשקים יכולים לקבל טיפוס גנרי, ואפשר לשמור על הטיפוסים הגנריים במחלקה הממשת.
- מגבילות:
 - לא ניתן לקרוא לבנייה של טיפוס גנרי.
 - הטיפוס הקונקרטי מוכרכ להיות טיפוס הפניה ולא פרימיטיבי (`Integer` ולא `int` למשל).

:טיפוסים נאים (raw types)

- עובד לנו עם **תאיות לאחרו לגורסאות ישנות של Java** שבהם לא היו טיפוסים גנריים. לא נשימוש בו בקוד, אלא רק בשאלות ב מבחון ☺
- יצירת טיפוס נא מתרחשת כאשר ניצור משתנה/מופע של מחלקה גנרית ונשミニ את הסוגרים המשולשים. עבור טיפוסים כאלה הקומPILEר לא מבצע בדיקות בטיחות טיפוסים. אפשר לבצע בדיקות טיפוסים raw לא שגיאת kompileaza.

```

public class Container<T> {
    private T val;
    public Container(T val) { this.val = val; }
    public T getVal() { return val; }
    public void setVal(T newVal) { val = newVal; }

    public static void main(String[] args) {
        Container<String> strCont = new Container<String>("Getting schwifty");
        strCont.setVal(0);      // Error – doesn't compile (which is good!)

        Container rawCont = new Container<String>("Getting schwifty");
        rawCont.setVal(0);      // No error (which is bad!)

        Container<String> rawCont2 = new Container(0); // No error (also bad)
        String s = rawCont2.getVal(); // Run-time error
    }                           class java.lang.Integer cannot be cast to class
                                java.lang.String

```

- diamond operator – כאשר מגדירים משתנה או שדה גנרי ומבצעים את ההשמה באותו שורה – אפשר להשתמש את הטיפוס הגנרי הקונקרטי אך להשאיר את הסוגרים המשולשים.

מתודות גנריות:

-
-

המחלקה לא חייבת להיות מוגדרת בGENERITY, רק כי המתוודה GENERITY.
אם למתוודה GENERITY יש טיפוס GENERY שחולק את אותו השם עם הטיפוס הGENERY של המחלקה בה היא נמצאת – זה אמן מבלבל, אך מדובר בשני טיפוסים לא קשורים. T שמוגדר במתוודה GENERITY הינו scope של הפונקציה הזאת. ה-T של המחלקה, הינו scope של המחלקה. רצוי להימנע מהבלבול ולקרא לAhead T ולשוני S לצורך העניין.

```
public class Helper<T> {
    public <T> boolean compare(Container<T> c1, Container<T> c2) {
        return c1.equals(c2);
    }
    public static void main(String[] args) {
        Helper<String> h = new Helper<>();
        Container<Integer> c1 = new Container<>(1);
        Container<Integer> c2 = new Container<>(2);
        h.compare(c1, c2); // this compiles
    }
}
```

מתודת סטטית GENERITY – מוכרכה להגדיר טיפוס משלמה. ברגע למתוודה מופיע שיקולו להגדיר טיפוס GENERY משלמה **או** להשתמש בטיפוס של המחלקה.

-

```
public class Helper<T> {
    public boolean compare(Container<T> c1, Container<T> c2) {...}
}
```

OK!

```
public class Helper<T> {
    public static boolean compare(Container<T> c1, Container<T> c2) {...}
```

Compilcation Error: cannot find symbol

```
public static <T> boolean compare(Container<T> c1, Container<T> c2) {...}
```

ירושה GENERITY:

-

אם כבר יודעים שנייתן לבצע את ההשמה הבאה:

```
String s = "IAmAnObjectToo";
Object o = s;
```

האם ההשמה זו חוקית?

```
Container<String> s = new Container<>("whoops");
Container<Object> o = s;
```

ההשמה אינה חוקית (**שגיית קומפליציה**) מכיוון שבאופן כללי, אם B מקיים `is-a` עם A, זה לא גורר שום `is-a` בין `GenericClass-ל-> GenericClass<A>`

A is a B, But `Container<String>` is not a `Container<Object>`
אין פה ירושה: `Container<Object>-ל-> Container<String>`

ג'וקרדים (wildcards)

- בקוד גנרי היסמן? מסמן **טיפוס לא ידוע**. כדי לעבוד איתם נctrיך להגדיר חסם עליון או תחתון. אם נקרא לפונקציה שמודיפה מספרים עם numbers <? extends Number>, הקומפיאילר לא יוכל לאפשר מעבר על רשימה מספרים כי אין כל הבטחה שמדובר ב-Number. נctrיך לקרוא לפונקציה עם Collection<? extends Number>.
- מוגבלות של חסמים:
 - **לא נוכל להוסיף אף איבר.** בשורה הראשונה נוכל לקבל רק List<Exception>. לא List<Exception> של כל מי שנמצא מתחתינו בהיררכיה. בغالל האופציה הזאת לא נוכל להכניס לרשימה הזאת איבר מטיפוס Exception.
 - **לעומת זאת במקרה של super נוכל להוסיף.**

• אילו שורות יעברו הידור?

```
public static void example(List<? extends Exception> lst){
     lst.add(new Exception("a"));
     Exception e = lst.remove(0);
}
```

• למעשה לא ניתן להוסיף אף איבר.

```
public static void example(List<? super Exception> lst){
     lst.add(new Exception("a"));
     Exception e = lst.remove(0);
     Object o = lst.remove(0);
}
```

• המגבלה תקפה גם אם הטיפוס הקונקרטי נכתב:

```
List<? extends Exception> lst = new ArrayList<Exception>();
lst.add(...);
```

• לא ניתן להוסיף אף איבר לרשימה!

- זאת מכיוון שלפי הטיפוס הסטטי הקומפיאילר לא יכול להיות בטוח שטיפוס האיבר שנוסף מקיים a-ו עם הטיפוס הקונקרטי הלא ידוע של הרשימה.

• שאלה מבחינה:

שאלה מבחינה

```
public class A<S> {
    public void f1(Collection<Number> l1, List<? extends Number> l2) {
        l1 = l2;
    }
    public <E> void f2(List<? super E> l, E elm) {
        l.add(elm);
    }
    public void f3(Collection <? extends S> c, List<S> l ) {
        c = l;
    }
}
```

א. רק f1 מתקاملות.
 ב. רק f2 מתקاملות.
 ג. רק f3 מתקاملות.
 ד. רק f1+f2 מתקاملות.
 ה. רק f1+f3 מתקاملות.
 ו. רק f2+f3 מתקاملות.
 ז. כל הפונקציות מתקاملות.
 ח. כל הפונקציות לא מתקاملות.

אילו מبنן הפונקציות f המופיעות במחלקה A מתקاملות? בחרו את התשובה הטובה ביותר.

העמסה והורשה

במקרים של העמסה, הקומפיילר מחליט איזה גרסה תרצו (או יותר נכון, איזה גרסה לא תרצו). נתבונן במקרים הבאים:

- אותו שם, פרטטר שונה – אם מדובר ב-boolean או double נתן להבחן לחוטין. אם מדובר ב- Rectangle ו- ColoredRectangle (שיורשת ממנה) אין לנו דרישת. **שתי הפונקציות יכולות להתקיים זו לצד זו** (העמסה). הקומפיילר יכול להחליט באופן הבא:

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

- מוגלם – להזכיר שני פרטטים בסדר שונה, פעם רגיל ואז צבוע, פעם צבוע ואז רגיל. כאשר יש קלט של צבע לצורך העניין, הוא יתאים לשתי הפונקציות, בשני המקרים נדרש המרה (בלפי מעלה). אין דרך להחליט, זה לא חוקי.

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

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

יותר גרעע:

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

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

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

מנגן הטעמה הוא סטטי: בוחר את החתימה של השירות (טיפוס העצם, שם השירות, מספר וסוג הפרמטרים), אבל עדין לא קובע איזה שירות ייקרא.

עבור הקראה (cr . overloaded(cr)) תבחר (בזמן קומפיילציה) החתימה:
B.overloaded(Rectangle)

בגלל שיעד הקראה הוא מטיפוס B השירות היחיד הרלבנטי הוא **האדום!**
בזמן ריצה מופעל מנגן השיגור הדינמי, שבודח בין השירותים בעלי חתימה זאת, את המתאים ביותר, לטיפוס הדינמי של יעד הקראה. הטיפוס הדינמי הוא S, لكن נבחר השירות **הירוק**.

כנ"ל אם הקראה היא: (cr . overloaded(cr))

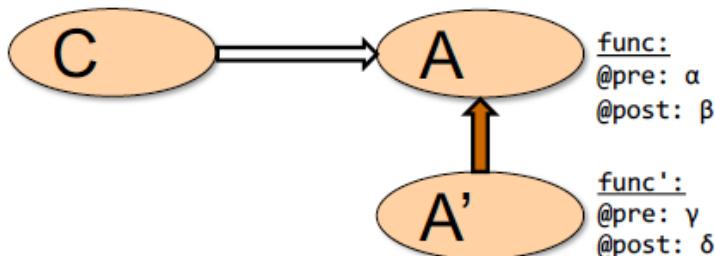


קבליות משנה – הורשה, טענות וחוזים

הורשה וטענות (assertions)

תנאי קדם, תנאי בתר ושמורות שהוגדרו עבור מחלקה או מנשך, תקפים גם לגבי עצמי המחלקה ומממשי המنشך, ועשויים להשתנות. עצם מהמחלקה נגזרת המוצבע ע"י הפניה מטיפוס מחלוקת הבסיס, צריך לקיים את השמורה שלה. מכאן **שמורה של כל מחלוקת צריכה להיות שווה או חזקה יותר משמרות הורשה**.

מחלקה C היא לקויה של מחלוקת A, כלומר יש ל-C הפניה ל-A באמצעות שדה, או שאחת המתוודות של C מקבלת פרמטר מסווג A. C מכירה את השמורה של A ומצפה מ-A לקיים אותה. בפועל, המצביע ל-A מצביע ל-'A, מחלוקת היורשת מ-A. ברור שבדי לקים יחסים פולימורפים תקינים, על 'A לפחות לפחות את שמרות A.



מחלקה 'A' דורשת שירות () z של A. מה יש לדרש מתנאי הקדם והבתר של השירות החדש ביחס לאלו של השירות המקורי?

- ראשית, אם ה-pre של A זהה ל-pre של 'A', וכן גם עם ה-post, הכל יהיה בסדר.
- **תנאי הקדם** – תנאי הקדם של 'A' צריך להיות לכל הפחות בתנאי הקדם של A (**שווה או חזק יותר**), כל קלט שאפשרי ב-A חייב להיות אפשרי ב-'A'. וכל להיות ש-'A' מאפשר יותר קליטים (אבל בטוח לא פחות).
- **תנאי הבתר** – תנאי הבתר של 'A' צריך **لتת לפחות את מה שניתן ב-A** (**שווה או חזק יותר**), יכול לתת גם יותר.

מבחינה לוגית:

- השמורה של מחלוקת יורשת צריכה להיות מרכיב מהשמורה של עצמה AND שמורה של כל מחלוקת שהיא יורשה ממנה לאורך עץ הירישה.
- **תנאי הקדם של מתודה שהוגדרה מחדש בחלוקת בלשוי**, הוא ה-OR של כל תנאי הקדם של מתודה זו בכל הוריה של המחלוקת לאורך העץ.
- **תנאי הבתר של מתודה שהוגדרה מחדש בחלוקת יורשת הוא AND**elogical של כל תנאי הבתר של הפונקציה אצל כל הוריה בעץ הירישה.

```

public class MathWizard {
    ...
    /** returns the square root of num
     * @pre epsilon >= 10 ^(-6)
     * @post abs($ret*$ret - num) <= epsilon
     */
    double sqrt(int num, double epsilon);
    ...
}
  
```

```

public class AccurateMathWizard extends MathWizard {
    ...
    /** returns the square root of num
     * @pre epsilon >= 10 ^(-20)
     * @post abs($ret*$ret - num) <= epsilon/2
     */
    double sqrt(int num, double epsilon);
    ...
}
  
```

בדוגמה תנאי הקדם חזק יותר (מורשה יותר ערכי אפסילון) ותנאי הבתר חזק יותר (מבטיח דיוק רב יותר)

שינויים בדיספה של פונקציות:

סוג	תיאור	למה לא עובד?
הורשה וחריגים	כלן משנה (מחלקה יורשת/מממשת) אימ' יכול לזרוק לאחרי הקלעים חריג שלא הוגדר בשירות הנדרס. למתוודה הדורסת מותר להקל על הליקון ולזרוק פחות חריגים מהמתוודה במחלקת הבסיס שלה. בדוגמה, EOFException הוא סוג של IOException ולכן זה אפשרי.	דוגמה: בהינתן מימוש המחלקה A, אילו מבין הגרסאות של func ניתן להוסיף למחלקה B שיורשת מ-A?
הורשה ונראות	למתוודה הדורסת מותר להקל את הנראות – כולמר להגדיר סטטוס נראות רחב יותר, אבל אסור להגדיר סטטוס מצומצם יותר. בדוגמה, באשר לא מגדרים נראות, הדיפוליטי היא package (ובוון-sh-protected מרחיב את package, זה לא אפשרי כאן).	<pre>public class A{ protected void func(){ } } public class B extends A{ <input checked="" type="checkbox"/> //public void func(){} <input checked="" type="checkbox"/> //protected void func(){} <input checked="" type="checkbox"/> //void func(){} <input checked="" type="checkbox"/> //private void func(){} }</pre>
הורשה והערך המוחזר	למתוודה הדורסת מותר לצמצם את טיפוס הערך המוחזר. בדוגמא, טיפוס הערך המוחזר הוא תת-טיפוס של טיפוס הערך המוחזר בתוודה במחלקת הבסיס שלה.	<pre>public class A{ public Number func() { return null; } } public class B extends A{ <input checked="" type="checkbox"/>//public Object func() { return null; } <input checked="" type="checkbox"/>//public Number func() { return null; } <input checked="" type="checkbox"/>//public Integer func() { return null; } }</pre>

סיכום (תרגיל 12)ממשקים:

- ממשק יכול להרחיב יותר ממנשך אחד.
- שירותים במנשך יכולים להיות: **פרטיים**, **ציבוריים**, ובברירת מחדל **מוספתיים**.
- שירות שלא נכתב עליו כלום הוא **בדיפולט public abstract**.
- שאלה: במנשך Foo מופיעעה פונקציה שזרוקת exception. אם נקרא לה ב-main ולא נזהיר/נטפל בשגיאה זה לא יתאפשר. אמן, אם דرسנו את המתוודה במימוש הממשק במחלקה **IfooImpl** ושם אנחנו לא זורקים שגיאה, **בשניצור אובייקט מהטיפוס IfooImpl** שדרס לא תהיה בעיה.
- כאשר אנו מימושים ממשק, אם לא נציין גראות לפונקציה, נקבל גראות שאינה **public**. לכן אם מדובר בפונקציה שמוגדרת במנשך כ-public, תהיהכאן חוסר התאמה, שגיאת קומpileציה.

Access Level

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	No modifier	Y	N	N
private	Y	N	N	N

חוקי דרישת:

- רשיימת הארגומנטים צריכה להיות זהה לו של המתודה הנדרשת.
- טיפוס ההחזרה צריך להיות זהה או subtype של טיפוס ההחזרה במתודה הנדרשת.
- נראות המתודה צריכה להיות זהה או רחבה יותר מזו של המתודה הנדרשת (אי אפשר לצמצם).
- אפשר לזרוק checked exception **צורות יתור**, או להוריד את הזריקה (אי אפשר להרchiיב).
- אי אפשר לדחוס מתודה שモוגדרת כ-final.
- אי אפשר לדחוס בנאום.
- מתודה סטטית לא יכולה להיות להידرس (כן אפשר להציג מחדש – hiding).**
- אם דרשו פונקציה private static בנסיבות אחרות והרטנו את ה-public (static למשל), אז במחלקה הדורשת יצרנו בפועל פונקציה חדשה, כי ה-private לא נורשת. אין כאן מצב של דרישת. אם היא הייתה protected, או protected יורשים אותה.

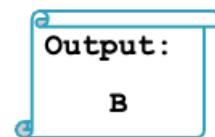
```

public class A {
    private static void foo() {
        System.out.println("A");
    }
}

public class B extends A {
    public void foo() {
        System.out.println("B");
    }
}

public class BindingTest {
    public static void main(String args[]) {
        B b = new B();
        b.foo();
    }
}

```



האם הקוד מתקין? אם לא, למה?
אחרת, מה הפלט?

:binding

סטטי – מתודות סטטיות, פרטיות, final, ושdot.

```

public class A {
    private void foo() {
        System.out.println("A.foo()");
    }

    public void bar() {
        System.out.println("A.bar()");
        foo();
    }
}

public class B extends A {
    public void foo() {
        System.out.println("B.foo()");
    }

    public static void main(String[] args)
    {
        A a = new B();
        a.bar();
    }
}

```

הפלט:
A.bar()
A.foo()

האם הקוד מתקין? אם לא, למה?
אם כן, האם יש שגיאת ריצה? אם יש, למה?
אחרת, מה הפלט?

סדר הפעולות ביצירת אובייקט:

- חשוב:

```
public class B extends A{
    String bar = "B.bar";
    B() { foo(); }
}
B(){
    bar = null;
    super();
    bar = "B.bar";
    foo();
}
```

סדר הפעולות ביצירת אובייקט

1. אתחל ערך דיפולטי לשדות מופע.
2. קריאה לבנאי של מחלקת האב (שגורר).
3. אתחל שדות מופע לפי הערכים שהושמו להם בשורה שבה הם מוגדרים.
4. ביצוע שאר הקוד של הבנאי.

- דוגמה מבבלת אך חשובה:

הורשה ובנאים

```
public class A {
    String bar = "A.bar";
    A() { foo(); }
    public void foo() {
        System.out.println("A.foo(): bar = " +
            bar);
    }
}

public class B extends A {
    String bar = "B.bar";
    B() { foo(); }
    public void foo() {
        System.out.println("B.foo(): bar = " +
            bar);
    }
}
```

```
public class C {
    public static void main(String[] args) {
        A a = new B();
        System.out.println("a.bar = " +
            + a.bar);
        a.foo();
    }
}
```

מה פלט התוכנית?

```
B.foo(): bar = null
B.foo(): bar = B.bar
a.bar = A.bar
B.foo(): bar = B.bar
```

- נשים לב שגם אם אנחנו לבנאי שמקבל ארגומנטים, עדין קריית ה-**super** היא לבנאי בירית המודול של האב. אם הוא לא קיים נקבל שגיאת קומpileציה!



העמסה:

- אפשר להגיד מספר שירותים עם אותו שם בתנאי **שטייפוס ערך החזרה / או מספר הארגומנטים** שונה.

דרישה והעמסה של שירותים

```
public class A {
    public float foo(float a, float b) throws IOException {
    }
}

public class B extends A {
    ...
}
```

אילו מהשירותים הבאים ניתן להגיד ב- B?

- 1. **float foo(float a, float b){...}**
- 2. **public int foo(int a, int b) throws Exception{...}**
- 3. **public float foo(float a, float b) throws Exception{...}**
- 4. **public float foo(float p, float q) {...}**

מחלקות פנימיות:

- מחלקת פנימית בתחום פונקציה, יכולה לגשת למשתנים מקומיים של הפונקציה רק אם הם **effectively final**.

מחלקות פנימיות

```
public class Test {
    public int a = 0;
    private int b = 1;
```

אילו משתנים מ- e-a נגישים מהשורה
המסומנת?

```
public void foo(final int c) {
    int d = 2;

    class InnerTest {
        private void bar(int e) {
            ...
        }
    }
    d = 3;
    a = 3;
}
```

תשובה: כולם חוץ מ-d

עבור **a** – יכול לגשת כי זה שדה מופיע והוא **Public**.
 עבור **d** – לאחר והיא מחלוקת פנימית יכול לגשת גם לשדה שהוא **private** של המחלוקת העוטפת שהם גם שדות מופיע. יכול לגשת גם לשדה שהוא **static** וגם **final**, כל מה הקשור למחלוקת. **שים לב! אם זו הייתה**

פונקציה סטטית לא יוכל לגשת לשדה מופיע.

עבור **c** – זה משתנה שהוא בפונקציה אליו נראה מוזר אבל המחלוקת היא בתחום הפונקציה והוא **final** אפשר לגשת אליו.

עבור **d** – לא כי הוא יכול להשתנות. מחלוקת לוקאלית וגם אוניברסלית יכולה לגשת עבור הפונקציה העוטפת רק לשדות שהם **final** או **effectively final**. פה **d** משתנה ולא **final** ולכן אין אפשרות לגשת. אולי מושנים את **d** פה רק אחרי אבל הוא לא **Effectively final** ולכן אין אפשרות לגשת אליו.
 זה ברמת הקומפיילר, הוא רץ על הקוד וידע אם המשתנה משתנה או לא.

עבור **e** – בתחום המחלוקת אין בעיה בכלל.



אוסףים גנריים

:HashSet

הכנסה ל-HashSet היא בתקווה מפוזרת בצורה אחידה על פנו כל התאים. כיוון שכל Point שנרצה להכניס הוא במקומם שונה בזיכרונו (אובייקט חדש בעל כתובת אחרת), אין שום קשר בין שני אובייקטים בעלי אותו תוכן. כדי לראות שאנו לא מכניסים את אותו איבר פעמיים, אנו דורסים את הפונקציה equals, אבל לא לשוכח לדرس גם את הפונקציה hashCode.

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

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

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

Output:
2



דרישות מהימוש של hashCode

- אין רנדומליות – עבור אותו האובייקט, hashCode צריכה להחזיר את אותו הערך בכל קריאה.
- אם שני אובייקטים מקיימים (x.equals(y) הפונקציה hashCode צריכה להחזיר את אותו הערך עבור שניהם.
- כדי לייצר ערכים שונים עבור אובייקטים שאינם מקיימים (x.equals(y) על מנת לשפר ביצועים.

:TreeSet

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

Output:
1



- אינו עובד עם hashCode (בשביל זה יש HashSet).
- אינו עובד עם equals (מפתחיע).
- השימוש של סט Comparator (תלוימם אם האלמנטים הם Comparable או משתמשים ב-Comparator), חייב להיות עקי עם equals.
- כיוון שהפונקציה compare יכולה באלה לדי משמעות באן, מה שרולונטי באן זה אך ורק ה-Comparator שהוא בפרמטר. לכן, הפונקציה equals אינה יכולה באלה לדי משמעות באן, מה שרולונטי באן זה אך ורק ה-Comparator שהוא בפרמטר. לכן, יכול לגלות בעיות שני אובייקטים שאינם equals, נחשבים כזהים ע"י ה-TreeSet.