תוכן עניינים

<u>הסיכום **לא** מכסה את הנושאים הבאים</u>:

- $HashSet$ – תרגול 4 ושבוע 6, Unit 6
- $Closure$ – שבוע 7, Unit 6, דקה 10:00
- $Garbage\ Collector$ (הרחבה) – תרגול 10 (הנושא האחרון)
- $Java\ Bytecode$ – תרגול 12 (הנושא האחרון)
- סביר מאוד שיש עוד נושאים שפספסתי.

הרצאות

שבוע 1:

https://www.youtube.com/watch?v=fWoTS-Qg5FY&list=PLFp170sgI_wLcEnnIOaLAqQxTChFZFjF6

שבוע 2:

https://www.youtube.com/watch?v=Ye2cTK5fQAI&list=PLFp170sgI_wLJr51fojhG-h_yS5Jv4o1b

שבוע 3:

https://www.youtube.com/watch?v=HG7pTlAMhsI&list=PLFp170sgI_wLNGcOiNIJqHsnsn6JXTq1y

שבוע 4:

https://www.youtube.com/watch?v=TPi4cV6ZRmg&list=PLFp170sgI_wJ6gBGGskwMOSM6LslqOlwj

שבוע 5:

https://www.youtube.com/watch?v=N76BqpYz5UY&list=PLFp170sgI_wLJCa-SXRHpEJVZwT6GQL_H

שבוע 6:

https://www.youtube.com/watch?v=n1rGqEktlNo&list=PLFp170sgI_wLXYb9tvboVjbT6NslALBKp

שבוע 7:

https://www.youtube.com/watch?v=g7D_fD1S_x8&list=PLFp170sgI_wJYSiVfboXNeKjloohbnRgk

שבוע 8:

https://www.youtube.com/watch?v=_fsLNg5VNK0&list=PLFp170sgI_wJqY9QibRzvm9zJmuCrafZN

שבוע 9:

https://www.youtube.com/watch?v=qEFbdvzJk6U&list=PLFp170sgI_wJwTT1j1SdDuTj0IvaK0fP5

שבוע 10:

https://www.youtube.com/watch?v=HRm19CClcfs&list=PLFp170sgI_wLnMAU6iCcyoEWoPF9T42OI

שבוע 11:

https://www.youtube.com/watch?v=wsa-EgvOrjc&list=PLFp170sgI_wK2qOnxAxpABq-fdMih1679

שבוע 12:

https://www.youtube.com/watch?v=2H1dv1iWe2E&list=PLFp170sgI_wJg5iOW6Qt2nXvh8cxJiian

שבוע 13:

https://www.youtube.com/watch?v=RRq5lYGavq4&list=PLFp170sgI_wIIjpid17xrLMx-620g9Vda

תרגולים

תרגולים 1-13:

https://drive.google.com/open?id=1O7zDsPthXg_M_iNLnq1_KRweY5QZ1zie

<u>טיפ</u>:

- מומלץ לעבור בעיקר על שבוע 13 וסיכום תרגול 13,
  שניהם עושים חזרה "זריזה" על כל החומר מתחילת הסמסטר.

רקע ומושגים בסיסיים בתכנות מונחה עצמים (OOP)

<u>מאפיינים של תוכנה טובה מצד המשתמש:</u>

- לעבוד.
- קל ללמוד אותה ולהשתמש בה.
- מהירה ויעילה.
- $Fail - Safe$ – לא קורסת, ללא באגים.
- $Fool - Safe$ – תדע להגן עלינו מטעויות של עצמנו (למשל ב-$Word$, אם נלחץ על כפתור היציאה X, התוכנה תשאל אותנו אם אנחנו רוצים לשמור את השינויים, למרות שבפועל היא יכלה גם לא לשאול וישר לצאת)
- $Hard - To - Hack$ – יודעת להגן מפני אנשים שמנסים לפגוע בתוכנית במכוון.
- $Compatible$ – אם נכתוב את הקוד לסביבה מסוימת, היא תהיה טובה גם לסביבות אחרות. למשל אם בנינו את התוכנה עבור $Windows$, היא תעבוד גם ב-$Linux$ וכו'. (שפת $Java$ היא $Compatible$)

<u>מאפיינים של תוכנה טובה מצד המתכנת:</u>

- קל/מהיר לקודד אותה.
- קל לבדוק אותה ולדבג אותה.
- קל להבין אותה. (קריאות וכו')
- קל לעשות בה שימוש חוזר.
- קל לעדכן אותה ולשדרג אותה.

<u>למה $Object - Oriented$?</u>

תכנות מונחה עצמים זו פרדיגמה שטובה למערכות תוכנה גדולות, כאשר התוכנה מכילה מרכיבים רבים שחולקים הרבה מהקוד אחד של השני, יש תלויות בין הרכיבים השונים ויש שינויים בדרישות המערכת, עולות דרישות חדשות וכו'. תכנות מונחה עצמים עומד בדרישות של תוכנה שקל להבין אותה, קל לבצע בה שימוש חוזר, וקל לעדכן ולשדרג אותה.

<u>מה זה תכנות מונחה עצמים?</u>

פרדיגמת תכנות שבה התוכנה מוגדרת כסט של אינטרקציות בין אובייקטים.

תכנות מונחה עצמים זו אלטרנטיבה לפרדיגמת תכנות אחרת - תכנות פרוצדוראלי.

בתכנות פרוצדוראלי, תוכנה היא רצף של פקודות,

ובתכנות מונחה עצמים, תוכנה היא רצף של פעולות על אובייקטים.

<u>$Objects$ - אובייקטים:</u>

אובייקטים בתוכנה יודעים להחזיק מידע ($data\ members$) ולבצע פעולות ($Methods$)

ההבדל בין תכנות פרוצדוראלי לתכנות מונחה עצמים:

נניח שיש לנו גן חיות, ואנחנו רוצים שכל חיה תוכל להשמיע קול.

בתכנות פרוצדוראלי, נבנה שיטות שיודעות לקבל חיה מסוימת ולגרום לה להשמיע קול:

$$bark(dog) \quad meow(cat) \quad moo(cow) \quad \ldots$$

כעת נבנה שיטה מסוימת שמקבלת חיה וגורמת לה להשמיע קול.

במקרה כזה, נצטרך לבדוק באיזה חיה מדובר ולשלוח את החיה לשיטה הרלוונטית,

מה שיגרור קוד ארוך מחולק לתנאים כדי לגרום לחיות להשמיע קול.

בתכנות מונחה עצמים, כל חיה תיוצג כאובייקט שמממש את המתודה $makeSound()$:

$$dog.makeSound() \quad cat.makeSound() \quad cow.makeSound() \quad \ldots$$

כעת אותה שיטה פשוט תצטרך להפעיל את השיטה $makeSound$,

בלי לדעת איזה חיה נשלחה לשיטה או איך נקראת השיטה של כל חיה.

אם נרצה להוסיף חיה חדשה, נייצר מחלקה חדשה שתממש את השיטה $makeSound$

והקוד ימשיך לפעול באותו אופן.

$Class$ – מחלקה:

יחידת תוכנה שמאפשרת לנו להגדיר קבוצה של אובייקטים,

שלכולם יש את אותם שדות ($data\ members$) ולכולם יש את אותן מתודות.

(מתודה = פונקציה שמוגדרת בתוך המחלקה ויכולה לגשת לשדות של המחלקה).

אובייקט של מחלקה מסוימת נקרא מופע של המחלקה ($instance$)

וכל מופע יכול לתת ערכים שונים לשדות שלו.

נשים לב שהחלק הפרוצדוראלי ב-$Java$ הוא המתודות,

כי החלק שבו אנחנו מריצים שורות קוד אחת אחרי השנייה מתבצע לרוב שם.

איך זה נראה בזיכרון:

- לכל מחלקה קיים עותק אחד בזיכרון של $Java$.
- לכל אובייקט (מופע של המחלקה) מוקצה גם מקום בזיכרון.
- כל אובייקט שייך בדיוק למחלקה אחת. (לא מדויק, נראה בהמשך הקורס)

Constructor

כדי ליצור אובייקטים (מופעים) של מחלקה מסוימת, נשתמש במתודה מיוחדת שנקראת בנאי.

דרך הבנאי נוכל לתת ערכים לשדות של האובייקט.

תכונות של בנאים:

- משתמשים באותו שם כמו המחלקה.
- לא מחזירים שום דבר (אין להם ערך החזרה, לא משתמשים במילה השמורה $return$)
- יכולים לקבל פרמטרים.

$\underline{Default\ Constructor}$:

לכל אובייקט ב-$Java$ קיים בנאי דיפולטיבי, כלומר בנאי שנמצא במחלקה מבלי שהגדרנו אותו בצורה מפורשת, והוא מאפשר לייצר מופעים של המחלקה. הבנאי יאתחל את כל השדות לערכם המחדלי ($int = 0, boolean = false, String = null$ וכן הלאה).

במידה והגדרנו בנאי כלשהו, הבנאי הדיפולטיבי **מתבטל** ולא ניתן יהיה לייצר מופע בעזרתו.

(נוכל כמובן להגדיר את הבנאי הדיפולטיבי באופן מפורש, כלומר בנאי שלא מקבל ערכים)

Overloading

ב-$Java$ נוכל להגדיר מספר שיטות עם אותו שם, בתנאי שמספר המשתנים / סדר המשתנים שונה.

מה הכוונה? בפיתון יכלנו להגדיר ערכים מחדליים בחתימת השיטה בעזרת " = ",

וכך בעצם הגדרנו שיטה אחת שיכולה לקבל מספר שונה של פרמטרים.

ב-$Java$ לא ניתן לעשות זאת באותה צורה, אלא בעזרת $Overloading$.

השתמשנו ב-$Overloading$ למשל בשיטה $System.out.println(value)$;

בכך שיכלנו לשלוח בתור $value$ משתנה מכל סוג שהוא ($int, double, String$ וכו')

איך בעצם זה אפשרי? מכיוון שמוגדרות מספר שיטות $println$ במחלקה out שכל אחת שונה מהשנייה בסוג הפרמטר שנשלח לשיטה.

this Keyword

אובייקט ב-$Java$ יכול להצביע לעצמו. בפיתון למשל השתמשנו ב-$self$ כדי לעשות זאת,

ב-$Java$ נשתמש במילה השמורה $this$. למה זה טוב?:

- כדי להתייחס למשתנים "מוצללים", למשל אם יש לנו שיטת $setName(String\ name)$; שמקבלת מחרוזת בשם $name$ אך גם שם השדה הוא $name$, איך נבצע את ההשמה? כך: $this.name = name$; כך $Java$ תדע להבדיל בין המשתנה של המחלקה לפרמטר.
- קריאה מפורשת לבנאי , למשל בתוך בנאי מסוים נוכל לעשות $this(param)$; כדי להשתמש בבנאי אחר שהגדרנו, שמקבל פרמטר אחד.
- כמשתנה בשיטה. נוכל לשלוח מצביע לעצמנו לשיטה מסוימת. למשל שיטה שמדפיסה רשימה של כל השיטות במחלקה $printAllMethods(this)$;.

Switch

עוזר במקרים בהם נצטרך לבצע פירוט ידני של מספר השוואות דומות. למשל:

```
char ch = 'b';
switch(ch) {
        case 'd':
                System.out.println("Case 1");
                break;
        case 'b': case 'c':
                System.out.println("Case 2");
                break;
        case 'x':
                System.out.println("Case 3");
                break;
 default:
                System.out.println("Default");
}
```

ניתן לשרשר תנאים, כפי שעשינו בתנאי השני (גם עבור $'b'$ וגם עבור $'c'$ יקרה אותו דבר)

מה שיקרה זה שבכל $case$ אנחנו נבדוק האם ch שווה (בעזרת $equals$) לתו הנתון,

ואם כן ניכנס ל-$case$. למשל עבור $case\ 'd'$, מתקיים $'b' \neq 'd'$ לכן לא נכנס לתנאי הראשון.

בתנאי השני מתקיים שוויון ולכן ניכנס אליו, נדפיס $Case\ 2$ ונסיים.

נשים לב שהסיבה היחידה שבגללה נסיים היא כי יש $break;$ מיד אחרי,

אחרת אנחנו פשוט נמשיך הלאה לתנאים הבאים **גם אם הם לא תואמים**. למשל בקוד הבא:

```
char ch = 'b';
switch(ch) {
        case 'd':
                System.out.println("Case 1");
                break;
        case 'b': case 'c':
                System.out.println("Case 2");
        case 'x':
                System.out.println("Case 3");
                break;
 default:
                System.out.println("Default");
}
```

אין $break$ ולכן יודפס $Case\ 2$ ולאחר מכן $Case\ 3$ ואז נסיים.

אם לא מתקיים שום $case$ ניכנס ל-$default$ ונדפיס $Default$,

אך אנחנו לא חייבים לממש את $default$.

<u>הערה</u>: $Switch$ יכול לקבל כקלט מחרוזת, תו, מספר שלם או $Enum$.

Type

כל משתנה ב-$Java$ הוא אחד מ-2 הבאים:

- $a\ \ reference$ (מצביע לאוביקט)

רפרנס / מצביע הוא לא אובייקט אמיתי, אלא משהו שמצביע לאובייקט בזיכרון.

לכל רפרנס יש סוג, שהוא השם של המחלקה: $Bicycle\ \ myBike$, $Dog\ myDog$ וכו'..

- $a\ \ primitive$ (פרימיטיבים ... $int, double, char,$)

כל הפרימיטיבים מאותו הסוג דורשים את אותה כמות של זיכרון, כלומר כל ה-intים ידרשו את אותה כמות זיכרון, כל ה-$double$ אותה כמות זיכרון וכן הלאה.

Possible Values	Name	Content	Size
'a', '4', '&'	char	A character	2 bytes
3, 2987	int	An integer	4 bytes
3000000000	long	An integer	8 bytes
2.25	float	A real number	4 bytes
98023422.83	double	A real number	8 bytes
true, false	boolean	True / False	Varies
[-128,127]	byte	Raw data	1 byte
32767	short	Raw data	2 bytes

<u>אופרטורים</u>:

$$a\ +=\ x; \quad \Leftrightarrow \quad a = a + x;$$
$$a\ -=\ x; \quad \Leftrightarrow \quad a = a - x;$$
$$a\ *=\ x; \quad \Leftrightarrow \quad a = a * x;$$
$$a\ \%=\ x; \quad \Leftrightarrow \quad a = a\%x;$$
$$a\ /=\ x; \quad \Leftrightarrow \quad a = a/x;$$

בנוסף קיימים הקיצורים הבאים:

$$n++; \quad \Leftrightarrow \quad n += 1;$$
$$++n; \quad \Leftrightarrow \quad n += 1;$$
$$n--; \quad \Leftrightarrow \quad n -= 1;$$
$$--n; \quad \Leftrightarrow \quad n -= 1;$$

<u>מה ההבדל אז בין $n++$ ל-$++n$ או בין $n--$ ל-$--n$</u>:

```
int x = 5;
int y = x++;
print(x + "," + y);
y = ++x;
print(x + "," + y);
```

בפעם הראשונה יודפס 6,5 ובפעם השנייה 7,7.

כלומר $int\ y = x++;$ קודם ביצע השמה של הערך של x ל-y ולאחר מכן הגדיל את x ב-1.

לעומת זאת, $y = ++x;$ קודם הגדיל את הערך של x ב-1, ולאחר מכן ביצע השמה ל-y.

Garbage Collector

"טיפוס" כלשהו שדואג לשחרר את הזיכרון במקרים בהם קיימים אובייקטים שאין בהם יותר שימוש.
נסתכל למשל על קטע הקוד הבא:

$Bicycle\ bike1 = new\ \ Bicycle(1);$

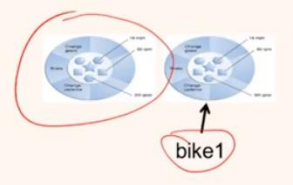
$bike1 = new\ Bicycle(1);$

$bike1 = new\ Bicycle(1);$

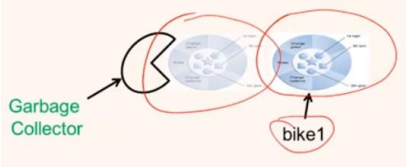
מה עשינו ואיך זה נראה בזיכרון?

יצרנו רפרנס בשם $bike1$ שמצביע לאובייקט $Bicycle(1)$ כלשהו:

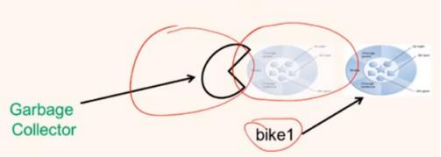
בשורה השנייה "דרסנו" את הרפרנס עם אובייקט חדש:



נשים לב שעכשיו אף אחד לא מצביע על האובייקט הראשון,
ולכן ה-$Garbage\ Collector$ "יאסוף" אותו (בשלב כלשהו)
ובכך יפנה מקום בזיכרון:



אחר-כך דרסנו את $bike1$ שוב,
ובאותו אופן אף אחד לא מצביע לאובייקט השני
לכן ה-$Garbage\ Collector$ "יאסוף" גם אותו
ויפנה מקום נוסף בזיכרון:



המחלקה $String$:

- המחלקה הכי נפוצה ב-$Java$. היא לא פרימיטיבית אך עדיין ניתן להגדיר אותה בעזרת הסימן " = " כמו שאפשר להגדיר פרימיטיבים. ($String\ myString = "hello";$)
- יש לה שיטות מעניינות כמו $length(),\ \ charAt()$ ועוד הרבה..
- היא $Immutable$, כלומר לא ניתן לשנות את **התוכן** של מחרוזת מסוימת.
(בכל פעם שנבצע שינוי במחרוזת, בפועל מה שנעשה זה להגדיר מחרוזת חדשה)
- למרות שהיא מתנהגת כמו $Primitive$, כדי להשוות מחרוזות נשתמש בשיטה $equals$ של המחלקה, ולא ב-" == ".

$Constants$ – קבועים:

שפות תכנות רבות (כולל $Java$) מאפשרות להגדיר משתנים קבועים, כלומר משתנים שלא ניתן לשנות את הערך שלהם בזמן הריצה. איך עושים את זה ב-$Java$? נשתמש במילה השמורה $final$ לפני סוג המשתנה, למשל: $final\ Bicycle\ myBike = new\ Bicycle(1);$

ובמקרה זה **לא נוכל** לשנות את המצביע $myBike$ שיצביע לאובייקט $Bicycle$ אחר.

באותו אופן עבור $final\ int\ myInt = 5;$, לא נוכל לשנות את ערכו של $myInt$ בזמן הריצה.

ההבדל בין $immutable$ ל-$final$

$final$ מונע שינוי של הרפרנס/מצביע עצמו,

אובייקט שהוא $immutable$ זה אובייקט שלא ניתן לשנות את התוכן שלו. כלומר:

אם נגדיר $String\ s = "hello";$ ואז $s = "hi";$ זה בסדר גמור (שינינו את המצביע).

לעומת זאת $s.charAt(0) = 'y';$ לא יתקמפל כי לא ניתן לשנות את **התוכן** של האובייקט.

באותו אופן, $final\ Bicycle\ myBike = new\ Bicycle(1);$ ואז $myBika.setSpeed(20);$

זה בסדר כי אפשר לשנות את התוכן של אובייקט $final$,

אך $myBike = new\ Bicycle(2);$ לא תקין כי אנחנו מנסים לשנות מצביע שהוא $final$.

למה למנוע מאחרים לבצע שינויים?

ניזכר במושג $fool - safe$ שבא למנוע מהמשתמשים לבצע טעויות.

במקרה הזה, אנחנו נגדיר כ-$final$ משתנים שלא אמורים להשתנות במהלך התוכנית,

ואם מישהו אכן מנסה לשנות אותם, זו כנראה טעות.

זה התפקיד שלנו בתור מתכנתים למנוע מדברים כאלה לקרות.

$Local\ Variables$ – משתנים לוקאליים:

- מוגדרים בתוך מתודה (בתחילת השיטה, בתוך $if/while$ וכו')
- בדומה לשדות ($data\ members$) גם להם יש סוג ($int, char, Dog$ וכו')
- יכולים להיות מוגדרים כקבועים ($final$)

$Scope$ - טווח:

כל קטע קוד שמונח בין סוגריים מסולסלים { }. למשל:

- תוכן של מחלקה $class\ MyClass\{\dots\}$
- מתודות $public\ static\ void\ main(String\ args[])\ \{\dots\}$
- לולאות, תנאים $if(\dots)\{\dots\}\ \ ,while(\dots)\{\dots\}$

ה-$Scope$ של משתנה, מגדיר מי יכול לגשת אליו או להשתמש בו.

לא ניתן לגשת למשתנים לוקאליים מ-$Scope$ חיצוני ל-$Scope$ בו הם הוגדרו.

כן ניתן לגשת אליהם מתוך $Scope$ פנימי יותר.

נראה דוגמה לקוד לא תקין:

```
if(...) {
        int internalNum = 5;
}
System.out.println(internalNum);
```

יגרור שגיאה כי ניסינו לגשת למשתנה $internalNum$ מחוץ ל-$Scope$ בו הוא הוגדר.

נראה דוגמה לקוד תקין:

```
int internalNum = 5;
if(...) {
        System.out.println(internalNum);
}
```

הקוד תקין כי אנחנו ניגשים למשתנה $internalNum$ שהוגדר ב-$Scope$ חיצוני.

$Namespace\ Pollution$:

הגדרת משתנים ב-$Scope$ הפנימי ביותר שניתן.

לא נרצה להגדיר משתנה ב-$Scope$ חיצוני מדי כי זה פוגע בקריאות וביכולת לתחזק ולעדכן.

יהיו מקרים בהם נעדיף להגדיר משתנים ב-$Scope$ חיצוני יותר, כדי לחסוך בזמן ריצה ומשאבים.

למשל בלולאות, נמנע מלהגדיר את אותו אובייקט שוב ושוב, ו נעדיף להגדיר אותו מחוץ ללולאה.

Static Modifier

המגדיר הסטטי מקשר בין משתנים או שיטות למחלקה ולא לאובייקט (אינסטנס ספציפי שלה).

Static Members – נשתמש במשתנה סטטי כדי לאחסן מידע שלא תלוי במופע ספציפי של המחלקה, אלא רלוונטי לכל המחלקה. משתנה שמוגדר כסטטי נקרא $Class\ Variable$ ומשתנה לא סטטי נקרא $Instance\ Variable$.

<u>הערה</u>: נניח שיש לנו משתנה סטטי בשם $nDogs$ במחלקה Dog שסופר את מספר הכלבים בעולם (כלומר מספר האובייקטים שייצרנו מהמחלקה הזו) כדי להשתמש במשתנה הזה,
נשתמש ב-$Dog.nDogs$ (כלומר שם המחלקה ואז שם המשתנה)
ולא $dog1.nDog$ (כאשר $dog1$ זה אינסטנס של המחלקה) כי המשתנה הוא משתנה של כל המחלקה ולא של אובייקט ספציפי, ולכן אין הגיון לפנות למשתנה סטטי של מופע ספציפי.

Static Methods – שיטות סטטיות לא פועלות על מופע ספציפי, אלא על כל המחלקה, לכן שיטות סטטיות לא יכולות לגשת לשדות ספציפיים של אובייקטים אלא רק למשתנים סטטיים של המחלקה.

<u>הערה</u>: איך נדע מתי להגדיר מתודה כסטטית ומתי לא?
אם המתודה מבצעת פעולות שלא קשורות למופעים ספציפיים של המחלקה,
למשל אם יש לנו משתנה סטטי $nDogs$ שסופר את מספר הכלבים שייצרנו עד עכשיו,
השיטה $getnDogs()$ (שמחזירה את הערך של המשתנה $nDogs$) תהיה סטטית.
כי היא לא צריכה לגשת לשום שדה של אינסטנס ספציפי, אלא למשתנה סטטי של המחלקה.
מנגד, אם המתודה מבצעת פעולות שקשורות לתוכן של שדות ספציפיים של המחלקה,
למשל השיטה $getDogName()$ (שמחזירה את השם של כלב ספציפי)
לא תהיה סטטית כי היא ניגשת לשדה של מופע ספציפי.

<u>מחלקה של מתודות סטטיות</u>:
יש מקרים בהם יהיה לנו אוסף של מתודות סטטיות שקשורות אחת לשנייה,
כלומר הן לא קשורות למופע ספציפי, ולכן נייצר מחלקה שתכיל את כולן.
הדוגמה הכי נפוצה היא המחלקה $Math$ שמכילה שיטות כמו abs, max, min וכו',
ואנחנו משתמשים בה בלי לייצר מופע כלשהו של המחלקה,
מכיוון שהיא מכילה מגוון של שיטות סטטיות שאינן קשורות למופע ספציפי.
נשים לב שבדומה לדרך בה אנחנו ניגשים למשתנים סטטיים,
גם כדי לגשת למתודות סטטיות אנחנו משתמשים בשם המחלקה, למשל $Math.abs(x)$ וכו'.

API

ה- API ($Application\ Programming\ Interface$) מגדיר את שער הכניסה לקוד שלנו, איך משתמשים בקוד שלנו, איזה משתנים יש, איזה מתודות וכו'.

$\underline{Minimal\ API}$:

תוכנות נוטות להיות יצורים יחסית מורכבים, אפילו תוכנות פשוטות יכולות להגיע לאלפי שורות של קוד, ובכך לגרום למי שמעוניין להשתמש בקוד שלנו להימנע מכך. לכן אנחנו שואפים לספק למשתמש כמה שיותר פונקציונליות עם כמה שפחות פרטים. כלומר $Minimal\ API$.
הרוב המוחלט של הקוד שלנו יהיה "חבוי" מהמשתמש, והוא ייחשף רק למידע מינימלי והכרחי.

למה לא לחלוק את הקוד שלנו במלואו?

- יותר זה פחות – ככל שיש יותר קוד, כך קשה יותר למשתמש ללמוד איך להשתמש בתוכנה.
- ככל שאנחנו מפרסמים יותר מידע על הקוד שלנו, כך קשה לנו יותר לשנות אותו בהמשך.
  כל פרט שאנחנו מספקים ללקוחות, הוא פרט שאנחנו מתחייבים עליו (מתודה למשל)
  ולכן הלקוח יכול לבנות על כך שהמתודה תמיד תהיה חלק מהתוכנית שלנו.
  אך אם נרצה למשל למחוק את המתודה בעתיד, לא נוכל לעשות זאת.

Casting

$cast$ זה התהליך של להמיר סוג מסוים לסוג אחר. יש 2 סוגי $cast$:

- $Excplicit\ casting$ (מפורש) בו אנחנו נרשום במפורש את הפקודה לביצוע $Cast$, למשל:
  $double\ a = 3.5;\qquad int\ b = (int)\ \ a;$
  (כלומר ביצענו $cast$ ל-a מ-$double$ ל-int בצורה מפורשת)
- $Implicit\ Casting$ (מרומז) בו $Java$ מבצעת את ה-$cast$ בעצמה, למשל:
  $double\ a = 3;$
  (כלומר $Java$ ביצעה $cast$ ל-3 מ-int ל-$double$)

Single Responsibility Principle

עקרון האחריות היחידה. לפי עיקרון זה, נשאף שלכל מחלקה תהיה אחריות אחת ויחידה,
כלומר לכל מחלקה יהיה תפקיד ספציפי שמשויך לה. למה?
מספר השינויים שנצטרך לבצע במחלקה שיש לה יותר מתפקיד אחד, יהיה כנראה גדול יותר מאשר מחלקה שיש לה רק תפקיד אחד. שינויים הם דברים שמועדים לבאגים, ובנוסף הם דורשים מאיתנו לבצע בדיקה חוזרת שהמחלקה עובדת כמו שצרך, מה שצורך כסף/זמן יקר.
(ככל שמחלקה גדולה יותר, כך קשה יותר לשנות אותה)

Information Hiding

אחד העקרונות המרכזיים בתכנות מונחה עצמים, מספק דרך פורמלית להגיע ל-$Minimal\ API$.

$\underline{Modifiers}$:

ב-$Java$ (כמו בשפות OO אחרות) נוכל להגדיר כל שדה וכל מתודה כ-$public$ או $private$.

$public$ – כל המחלקות יכולות לגשת לשדה / למתודה שמוגדר כ-$public$

$private$ – מונע גישה ממחלקות **אחרות** שמנסות לגשת לשדה / למתודה (ניסיון כזה יגרור שגיאה)

לרוב נרצה שכל השדות ($data\ members$) שלנו יהיו $private$.

<u>הערה</u>: לקוחות רואים רק את ה-$Public\ API$. (בהמשך נדבר על $modifiers$ נוספים)

$\underline{Getters\ /\ Setters}$:

שיטות שמוגדרות כ-$Public$ ומאפשרות גישה עקיפה למשתנים שהגדרנו כ-$private$.

למשל $getName()$ (לקבל את המשתנה $name$) או $setName()$ (לשנות את המשתנה $name$)

למה להשתמש ב-$Getters\ /\ Setters$ ולא להגדיר את המשתנים כ-$Public$?

- נוכל למנוע גישה ישירה למשתנים, ובכך למנוע למשל שינוי לא תקין שלהם.
- נוכל לא להגדיר בכלל שיטות $Setters$ ובכך למנוע לגמרי שינוי של המשתנים.
- נוכל לשנות את שמות המשתנים בהמשך, מבלי לשנות את ה-API.

  למשל משתנה $String$ שנקרא $name$ ואנחנו רוצים לשנות אותו למערך של $Char$.

  לא נצטרך לעדכן את ה-API אלא רק את השיטות $getName()$ ו-$setName()$ לעבוד בדרך החדשה שהגדרנו. מבחינת המשתמש, לא בוצע שום שינוי.

$\underline{Encapsulation}$ - <u>כימוס</u>:

הרעיון של כימוס הוא לקחת קבוצה של רעיונות ולאחד אותם תחת יחידה אחת עם שם אחד.

המטרה של כימוס היא לחסוך זיכרון של מחשב ויותר מכך זיכרון של בני אדם על ידי לקחת נושא מורכב ולהתייחס אליו כמשהו הרבה יותר פשוט.

נסתכל למשל על כפתור $On\backslash Off$. כמעט לכל מכשיר שאנחנו משתמשים בו ביום-יום יש כפתור $On\backslash Off$, שמאחורי הקלעים מבצע מספר פעולות, לעיתים מורכבות, כדי להפעיל את המכשיר, אך מבחינתנו כל מה שעשינו זה ללחוץ על הכפתור.

כימוס מתקשר באופן ישיר ל-$Information\ Hiding$, שכן אנחנו מסתירים מידע לא רלוונטי וחושפים מעט ככל שניתן.

Inheritance

בין 2 מחלקות שונות אפשר לחשוב על הרבה מערכות יחסים,
וב-OOP נוכל להגדיר את היחסים האלה בקוד שלנו. נסתכל על סוגים שונים של מערכות יחסים:

<u>$Has - a\ Relation$</u>: היחס הבסיסי ביותר בין מחלקות. (נקרא גם $composition$ - הכלה)
כלומר למחלקה א' יש רכיב מסוג מחלקה ב'. היחס בא לידי ביטוי כאשר מחלקה מסוימת שייכת
למחלקה אחרת, למשל: $Person\ has\ a\ name$, $Bicycles\ have\ wheels$ וכו'.
ב-$Java$, יחס של $has - a$ ממומש בעזרת $data\ members$ (שדות).

<u>$Is - a\ Relation$</u>: כאשר מחלקה א' היא סוג של מחלקה ב', למשל $Student\ is\ a\ Person$.
סטודנט חולק מספר דברים משותפים עם אנשים אחרים כמו למשל דיבור, הליכה וכו',
מצד שני לסטודנטים יש סט של יכולות שאדם כללי לא יכול לעשות כמו למשל לקחת מבחנים.
ב-$Java$, יחס של $Is - a$ בא לידי ביטוי על ידי ירושה ($Inheritance$)
נגיד ש-$A\ extends\ B$ אם A הוא סוג של B.
A הוא ה-$subclass$ של B ו-B הוא ה-$superclass$ של A.
למחלקה A יהיו כל המתודות, הבנאים והשדות של מחלקה B
אך היא יכולה להוסיף גם יכולות משל עצמה.

<u>$Instance - of\ Relation$</u>: נשים לב שלא להתבלבל בין היחס הזה ליחס $Is - a$.
גם על היחס הזה נוכל לחשוב כסוג של $Is - a$ למשל $Pluto\ is\ a\ Dog$,
אבל נשים לב שפלוטו הוא לא סוג ($Type$) של כלב , אלא ממש כלב ספציפי , מופע של כלב.
ב-$Java$, יחס של $Instance - of$ ממומש (איך לא) על ידי יצירת $instance$ ($new\ Dog()$)

<u>תכונות של ירושה</u>:

- ירושה היא רקורסיבית (כלומר מחלקה A יכולה לרשת ממחלקה B שיורשת ממחלקה C וכו')
- ירושה היא טרנזיטיבית (אם A יורשת מ-B ו-B יורשת מ-C אז A יורשת מ-C)
- כל מחלקה יכולה להיות מחלקת אב ($superclass$) של מספר בלתי מוגבל של מחלקות.
- מצד שני, כל מחלקה יכולה להיות מחלקת בן ($subclass$) של מחלקה אחת בלבד.
  (אם לא ציינו בצורה מפורשת ממי יורשת המחלקה, היא תירש מ-$Object$ כברירת מחדל)
- מחלקה יורשת לא יכולה לגשת לשדות / מתודות $private$ שמוגדרות במחלקת האב שלה.

<u>המחלקה $Object$</u>:

- המחלקה היחידה ב-$Java$ שאין לה מחלקת אב.
- יש לה מתודות שימושיות כגון $toString(), equals(Object\ other)$ ועוד..

Protected Modifier

בדומה ל-$public$ ו-$private$, מתודות / שדות / בנאים יכולים להשתמש במגדיר $protected$,
אשר מאפשר למחלקות יורשות לקבל גישה אליהם (להבדיל מ-$private$)
ומצד שני, מי שלא יורש מהמחלקה לא יכול לגשת אליהם (להבדיל מ-$public$)
לרוב, נשתדל להימנע מלהשתמש ב-$protected$ כי כל מה שמוגדר כ-$protected$ מתווסף ל-API.
למה זה רע? מאותן סיבות שהסברנו מקודם לגבי ה-$Minimal\ API$.
(בהמשך הקורס נכיר $Modifier$ נוסף שיחליף לרוב את השימוש ב-$Protected$)

Overriding

(לא מדובר ב-$Overloading$, לא להתבלבל)
$Overriding$ זה הרעיון של לרשת ממחלקה כלשהי ולשנות את ההתנהגות שלה.
למשל לקחת מתודה שמוגדרת כ-$Public$ או $Protected$ ולדרוס אותה, להגדיר אותה באופן שונה.
איך נממש את זה ב-$Java$? פשוט נגדיר את המתודה (עם אותה חתימה בדיוק) במחלקה שלנו,
ובכך נדרוס את המתודה של מחלקה האב.

<u>המילה השמורה $super$</u>:
מאפשרת לנו לגשת למתודה של מחלקה האב, למשל במקרים בהם נרצה להוסיף פונקציונליות
למתודה במחלקת האב, ולא לדרוס אותה לגמרי. איך נבצע את זה?
נוכל לקרוא למתודה של מחלקה האב בעזרת $super.methodName()$,
כלומר המילה השמורה $super$ היא מצביע למחלקת האב.
באותו אופן, נוכל לפנות לבנאי של מחלקת האב מתוך הבנאי שלנו בעזרת $super(arg1, arg2, \dots)$

<u>הערה חשובה:</u> כדי להגדיר אובייקט מסוג $Student$ (למשל), $Java$ צריכה להגדיר קודם כל אובייקט
מסוג $Person$, ולפני זה אובייקט מסוג $Object$. לכן בכל פעם שמתבצעת קריאה לבנאי של
$Student$, באופן דיפולטיבי, גם אם אנחנו לא כותבים $super(arge1 \dots)$ באופן מפורש, $Java$ פונה
קודם כל לבנאי הדיפולטיבי של $Person$, ומתוך הבנאי של $Person$ היא פונה לבנאי של $Object$.
זו הסיבה שהקריאה ל-$Super(arg1..)$ מתוך הבנאי של $Student$ חייבת להיות השורה הראשונה
בבנאי, אחרת זה לא יתקמפל. בנוסף, אם למחלקת האב שלנו אין בנאי **דיפולטיבי** (כזה שלא מקבל
ערכים בכלל), אנחנו נהיה חייבים להוסיף קריאה מפורשת ל-$super(arg1, \dots)$, כי $Java$ לא תדע
איך לפנות לבנאי שכן קיים. אם לא נעשה זאת, תהיה שגיאת קומפילציה.

<u>למה להשתמש בירושה?</u>
ירושה מונעת שכפול קוד, ולכן נשאף להשתמש בירושה לכל 2 מחלקות שמקיימות את היחס $Is-a$.
אם המחלקות לא מקיימות את היחס, **לא** נגדיר אותן בעזרת ירושה.
קיימות מספר דרכים נוספות למניעת שכפול קוד שנלמד בהמשך.
בנוסף, ירושה זו הדרך העיקרית למימוש פולימורפיזם ב-$Java$.

Polymorphism

פולימורפיזם (רב־צורתיות) הוא תכונה של שפות תכנות המאפשרת לטפל בערכים מטיפוסים שונים בעזרת ממשק תוכנה אחיד. פולמירופיזם הוא עיקרון בסיסי וחשוב בתכנות מונחה עצמים שעומד בבסיס ומתקשר עם הרבה עקרונות אחרים כמו אינקפסולציה, ירושה, מודולריות וכו'. פולימורפיזם ב-OOP מתקשר ליכולת של אובייקטים מסוימים לקחת לעצמם צורות שונות (כלומר הרחבת מחלקות).

נניח שיש לנו מחלקות Dog, Cow ו-$Animal$. המחלקות Dog ו-Cow יורשות מ-$Animal$,
המחלקות Dog ו-Cow מממשות את השיטה $speak()$
וגם המחלקה $Animal$ מממשת את $speak()$. נסתכל על הקוד הבא:

```
public void makeAnimalsSpeak(Animal[] animals) {
        for(Animal animal : animals) {
                animal.speak();
        }
}
```

- קיבלנו בחתימת השיטה מערך של $Animal$, עברנו על כל חיה במערך והפעלנו את $speak$.
  כלומר, מבלי לדעת איזה חיות יש במערך, יכלנו לגרום לכל חיה להשמיע קול.
  זה הרעיון הכללי של פולימורפיזם, לדעת לטפל באובייקטים מסוגים שונים, תוך הנחות מינימליות עליהן כמו במקרה שלנו שהן ממשות את $speak$.
- נשים לב שהשיטה שתרוץ בפועל זו השיטה שהוגדרה ב-Cow או ב-Dog,
  אלא אם כן האובייקט במערך הוא אובייקט מסוג $Animal$, ואז השיטה ב-$Animal$ תרוץ.
  אם השיטה $speak$ לא הייתה מוגדרת גם במחלקה $Animal$, הקוד לא היה מתקמפל.

<u>$Shadowing$ - הצלה</u>:

ראינו בדוגמה לעיל שלמשל עבור $animal.speak()$ השיטה שתרוץ בפועל זו השיטה של האובייקט עצמו ($Dog, Cow, Animal$) אך מה שקובע אם בכלל הקוד יתקמפל במקרה הזה הוא שהמחלקה של הרפרנס ($Animal$) מימשה גם היא את השיטה $speak$.
הדבר הזה לא נכון לגבי שדות ומתודות סטטיות, במקרים האלה רק הרפרנס קובע.

בדוגמה הבאה מודפס:

1

A

למרות שהיינו מצפים שיודפס:

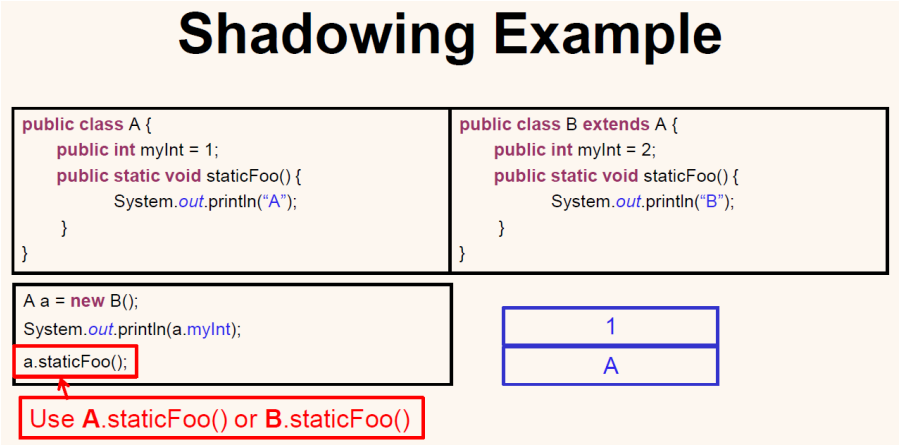
2

B

לכן נוכל להשתמש ב:

$A.staticFoo()$

או $B.staticFoo()$



$\underline{Polymorphism\ and\ Extensibility}$:

לפולימורפיזם יש קשר ישיר ליכולת להרחיב תוכנה. למשל בדוגמה הקודמת,
נוכל פשוט לייצר מחלקה שמייצגת חיה נוספת, למשל $Goat$, ולממש אצלה את $Speak$,
כל שהקוד ימשיך לעבוד כרגיל.

$\underline{Polymorphism\ and\ Flexibility}$:

לפולימורפיזם יש גם קשר ישיר לשינוי התנהגות בזמן ריצה.
למשל אם יש לנו $Animal\ myAnimal = new\ cow()$ ובזמן ריצה (נניח לפי קלט מהמשתמש)
אנחנו רוצים שהמשתנה יהיה כלב ולא פרה, אין עם זה שום בעיה, פשוט נעשה:
$myAnimal = new\ Dog()$ מה שלא ניתן ללא פולימורפיזם.

$\underline{Polymorphism\ and\ Minimal\ API}$:

כשדיברנו על $Minimal\ API$ אמרנו שהשאיפה היא להגדיר כמה שיותר משתנים ומתודות
כ-$private$, ולחשוף בפני המשתמשים כמה שפחות. בעזרת פולימורפיזם נוכל לקחת את זה אפילו
צעד אחד קדימה, ולהשתמש בעיקרון "תכנות לממשק ולא למימוש".
לפי העיקרון הזה, נשאף להגדיר את המתודות והמשתנים גבוה ככל הניתן בהיררכיה.
למשל אם נסתכל בדוגמאות הקודמות, נשאף להגדיר את $speak$ ב-$Animal$ ולא רק ב-Cow וכו'.

- הקוד באופן הזה הוא כללי יותר ($Generality$)
- ניתן להרחבה בקלות ($Extensibility$)
- מאפשר ללקוחות לא להיות מודעים לסוג האובייקט שאנחנו עובדים איתו.
  למשל בדוגמאות הקודמות, הלקוחות יודעים שאנחנו עובדים עם $Animal$
  אבל לא יודעים עם איזה חיה ספציפית נעבוד כמו למשל Cow או Dog.
  (מה שמאפשר בעתיד לבצע שינויים יותר בקלות ולהיות פחות תלויים במימוש ספציפי)
- יותר קל ללמוד אותו. המשתמשים מודעים לפחות סוגים שונים של מחלקות,
  נחשפים רק למחלקות הגבוהות ביותר בהיררכיה ובכך נחשפים לפחות קוד שצריך ללמוד.

Arrays

מערך ב-$Java$ הוא אובייקט מיוחד, אוסף (בגודל **קבוע**) של מספר אובייקטים מסוג **ספציפי**.

ניתן ליצור מערך במספר דרכים:

- בדרך הרגילה והמוכרת: $String[]\ strArray\ = new\ String[2];$

  (הרפרנס $strArray$ מצביע למערך בזיכרון שמכיל 2 מקומות)

  נוכל להוסיף איברים בעזרת: $strArray[1] = "Jack";$ ו-$strArray[0] = "Dana";$

- בדרך מקוצרת: $String[]\ strArray\ = new\ String[]\{"Dana","Jack"\};$

  יצרנו את המערך עם הערכים בתוכו. ניתן להשתמש בסינטקס הזה גם אחרי ההצהרה:

  $strArray = new\ String[]\{"Jack","Dana"\};$

- בדרך יותר מקוצרת: $int[]\ intsArray = \{32,64\};$

  הסינטקס הזה חוקי אך ורק בזמן ההצהרה, כלומר $intsArray = \{32,64\};$ לא חוקי.

- נוכל להגדיר גם מערך שמכיל מערכים $int[][]\ intsArray\ = \{\{1,2\},\{4,8\},\{1,2,3\}\}$

Foreach

דרך קלה לבצע איטרציה על מערכים או על איטרטורים בכללי. הסינטקס הוא כזה:

```
for (type var : array) {
    statements using var;
}
```

כאשר $type$ הוא סוג האובייקט שנמצא בתוך המערך, var יהיה האובייקט הנוכחי בריצה,

ו-$array$ הוא המערך שעליו נרוץ. נראה דוגמה:

```
for(int i : intsArray) {
    System.out.println("An element in the list: " + i);
}
```

כלומר רצנו על המערך $intsArray$, כאשר כל ערך במערך הוא מסוג int,

והערך הנוכחי מיוצג באמצעות i בלולאה עצמה.

<u>הערה</u>: לולאת $foreach$ לא מבצעת מעקב על האינדקסים , כך שלא ניתן לדעת באיזה אינדקס במערך נמצא var. (אפשר כמובן להגדיר משתנה $counter$ שיתעדכן כל איטרציה)

Primitive Wrappers

מחלקות ב-$Java$ שעוטפות פרימיטיבים. למה להשתמש בהם?

- כדי שנוכל להשתמש בשיטות שדורשים אובייקט כקלט (למשל $Collections$)
- כדי להשתמש בקבועים שמוגדרים במחלקות האלה. (למשל $Integer.MAX\_VALUE$)
- כדי להשתמש בשיטות שימושיות שמומשו במחלקות האלה (בדרך כלל להמרות)

  למשל: $Integer.parseInt("5");$ או $Double.parseDouble("5.0");$

  או למשל עבור אובייקט $Integer$ כלשהו: $Integer\ myInt = new\ Integer(5);$

  נוכל להפוך אותו לפרימיטיבי על ידי: $int\ primitve = myInt.intValue();$

Abstract Classes

מחלקות אבסטרקטיות הן מחלקות רגילות שיש לפניהן את המילה השמורה $abstract$,
ואין אפשרות ליצור מופעים ($instances$) חדשים שלהם. (השימוש ב-new יגרור שגיאת קומפילציה)
מתי נגדיר מחלקה להיות אבסטרקטית? כשאין משמעות ליצור מופעים של המחלקה.
למשל בדוגמאות הקודמות, אין משמעות לייצר מופע של $Animal$, כי מדובר בחיות ספציפיות כמו
למשל Cow או Dog ולכן נייצר מופעים של המחלקות האלה ולא של $Animal$.
בנוסף, $Animal$ מכילה מימושים דיפולטיביים לחלק מהשיטות והשדות שמשותפים לכל החיות.
במקרה הזה, נגדיר את $Animal$ להיות מחלקה אבסטרקטית.

<u>מאפיינים של מחלקות אבסטרקטיות</u>:

- ניתן להגדיר שיטות להיות אבסטרקטיות, כלומר שיטות שאין להם מימוש במחלקה עצמה אך כל מחלקה (לא אבסטרקטית) שיורשת ממנה חייבת לממש את המתודה, אחרת היא לא תעבור קומפילציה. (אם המחלקה היורשת גם אבסטרקטית, היא יכולה לממש אותה או לא)
- מחלקה אבסטרקטית יכולה להחזיק מתודות ושדות כמו כל מחלקה אחרת.
- מתודות סטטיות לא יכולות להיות מוגדרות כ-$abstract$.
- ניסיון לקרוא לשיטה אבסטרקטית בעזרת $super$ (כלומר שמחלקת בן תקרא לשיטה אבסטרקטית שנמצאת במחלקת האב) יגרור שגיאת קומפילציה.
- שיטות אבסטרקטיות לא יכולות להיות $private$,
  יש בזה היגיון כי אם הן כן יהיו $private$, מחלקת הבן לא תוכל לדרוס אותה.

Interfaces

סוג של מחלקה, שיכולה להכיל רק 2 דברים: קבועים ומתודות אבסטרקטיות.
בדומה למחלקה אבסטרקטית, לא ניתן לייצר מופעים של מחלקות שמוגדרות כ-$interface$.
מה שכן ניתן לעשות זה לממש אותן או לרשת מהן (רק $interface$ יכול לרשת מ-$interface$)
איך זה נראה ב-$Java$? נשתמש במילה השמורה $interface$ בהגדרת המחלקה,
למשל $public\ interface\ Printable$.

<u>למה להשתמש ב-$interface$?</u>

- כדי להגדיר "חוזים" / "הסכמים" שמחלקות לוקחות על עצמן, למשל:
  $Printable$: אינטרפייס שמציין שניתן להדפיס אובייקט מסוים.
  $Comparable$: אינטרפייס שמציין שניתן להשוות בין האובייקט לבין דברים אחרים.
  כל מחלקה שמממשת את אחד מהאינטרפייסים הנ"ל "מודיעה" לנו שיש לה את היכולות הנדרשות להדפסה / השוואה כנדרש. המחלקה מצידה נדרשת לממש את כל המתודות האבסטרקטיות באינטרפייס.
- כדי להגדיר API קבוע. נניח שיש לנו מספר מחלקות שונות שצריכות לממש את אותו API, נוכל להגדיר אינטרפייס שמכיל חתימות אבסטרקטיות של השיטות הנדרשות, ולוודא שכל המחלקות מממשות את האינטרפייס הזה. בכך נבטיח שלכולם יהיה את אותו API.

<u>$Interfaces\ and\ modifiers$</u>:

בהמשך למה שאמרנו לעיל, בגלל שאינטרפייס מגדיר API מסוים,
ניתן להגדיר בו אך ורק מתודות $Public$.
באותו אופן לא ניתן להגדיר שדות פרטיים אלא רק שדות $final\ static$ (קבועים)

<u>$Default\ Methods$</u>:

$Interface$ יכול להוסיף גם מתודות דיפולטיביות. מה הכוונה?
מי שיממש את האינטרפייס, לא מחויב לממש את השיטות הדיפולטיביות. אז למה זה טוב?

- כדי להוסיף שיטות שיכולות להיות שימושיות, אך לא הכרחיות.
- כדי להוסיף שיטות שרלוונטיות רק למקרים מסוימים ("לסגור חורים")

הערה חשובה: כשדיברנו על ירושה, אמרנו שכל מחלקה יכולה לרשת ($extends$) רק מחלקה אחת.
כעת במקרה של $Interfaces$, מחלקה יכולה לממש ($implements$) כמה אינטרפייסים שהיא רוצה.
לכן, לכל מחלקה יכולים להיות כמה $Type$ים שונים:

- ה-$Type$ של המחלקה עצמה
- ה-$Type$ של המחלקה ממנה היא יורשת (וכל מחלקה גבוהה יותר בהיררכיה)
- ה-$Type$ים שהיא מממשת

ולכן אנחנו חוזרים שוב לעניין של פולימורפיזם, כאשר ניתן להסתכל על כל מחלקה לפי כל אחד מה-$Type$ים שתיארנו לעיל, למשל נוכל ליצור מערך של אובייקטים שמממשים את $Printable$,
וכך מבלי לדעת שום דבר מעבר על האובייקטים עצמם, נוכל להדפיס כל אובייקט במערך.
(במקרה הזה, לא נוכל לבצע פעולות שלא מוגדרות באינטרפייס $Printable$ על אף אובייקט במערך,
בדיוק מהסיבה שאנחנו לא יודעים על האובייקט שום דבר מעבר לכך שהוא מממש את $Printable$)

Abstract class, Normal class and Interface

ניסיון (לא הכי מוצלח) לעשות סדר בין סוגים שונים של מחלקות.
איך נדע לבחור בין מחלקה אבסטרקטית, מחלקה רגילה או אינטרפייס?
נגדיר את היחס בין 2 מחלקות A, B באופן הבא:

האם A ו-B הם 2 סוגים של אותו דבר?

1. לא ⇐ $Interface$
2. כן
   א. האם יש להם מימוש משותף?
      i. לא ⇐ $Interface$
3. לא בטוח
   א. האם יש להם מימוש במשותף?
      i. חלק ⇐ $Interface$ עם שיטות $default$
      ii. לא בטוח
         a. האם המימוש המשותף דורש שדות?
            i. כן ⇐ $Interface\ and\ composition$
      iii. במידה רבה
         a. האם הגיוני שמחלקת האב תממש את השיטות?
            i. לא בטוח ⇐ $Inheritance\ with\ Abstract\ class$
            ii. כן
         1. האם הגיוני ליצור אינסטנס של מחלקת האב?
            a. לא ⇐ $Inheritance\ with\ Abstract\ class$
            b. כן ⇐ מחלקה רגילה

Reuse Mechanisms

כתיבת קוד שניתן למחזור בקלות. 2 דרכים נפוצות לכך:

נניח שאנחנו מעוניינים למחזר קוד מהמחלקה A למחלקה שאנחנו כותבים B:

<u>ירושה</u> – B יורשת את A ולכן יכולה להשתמש במתודות שלה ובכך "ממחזרת" את הקוד.

<u>$Composition$ (הרכבה)</u> - נגדיר אובייקט של A כשדה ב-B וכך נוכל להשתמש בשיטות של A.

<u>יתרונות של ירושה:</u>

- ברורה לשימוש. חלק מהמנגנון של השפה.
- מאפשרת פולימורפיזם.
- מוגדרת באופן סטטי בזמן הקומפילציה (לא משתנה בזמן הריצה ולכן בטוחה לשימוש)
- קל לשנות את מימוש הקוד על ידי $overriding$

<u>חסרונות לירושה:</u>

- הוגדר בתור יתרון אך הוא גם חסרון: לא יכול להשתנות בזמן הריצה (במקרים שכן נרצה)
- נוגד לעיתים את עקרון האינקפסולציה (חושף בפנינו מתודות נוספות ופרטים נוספים)
- המחלקה היורשת כפופה למימוש של האבא.
- אפשר לרשת רק מחלקה אחת.

<u>יתרונות של $Composition$:</u>

- מוגדר דינמי בזמן הריצה (אם נרצה, נוכל להחליף את האובייקט באובייקט מתאים יותר)
- שומר על האינקפסולציה (אנחנו לא חשופים באופן ישיר לשיטות ולמידע של האובייקט)
- חוסר תלות בין האובייקט לבין המחלקה (האובייקט לא כופה עלינו שום דבר)
- שומר על הרעיון שלכל מחלקה יש משימה אחת לדאוג לה.
- מחלקה יכולה להחזיק מספר בלתי מוגבל של אובייקטים.

<u>חסרונות של $Composition$:</u>

- יש לו יותר אובייקטים ולכן יהיה פחות מובן (מצד שני יש לו פחות מחלקות)
- לא ניתן להשתמש בפולימורפיזם.
- פוטנציאל לטעויות כי המבנה מורכב יותר.

- נשתמש בירושה אם "$A\ is\ a\ B$" (למשל כלב הוא חיה) אחרת, נשתמש ב-$Compositin$.
- $Compositin$ זו הדרך הנכונה לבצע שחזור קוד, לכן במקרים של שחזור קוד בלבד אשר התנאי הקודם לא מתקיים, נשתמש ב-$Composition$.

Casting

(נניח לצורך ההסבר כי Cow יורשת מ-$Animal$)

$\underline{Up - Casting}$:

"שדרוג אובייקט" – למשל להפוך את Cow ל-$Animal$.

דוגמה 1: $Animal\ a = new\ Cow();$

דוגמה 2: $Animal\ myAnimal = (Animal)\ new\ Dog();$

כאשר צד שמאל (הרפרנס) זו מחלקת אב (או $Interface$) של האובייקט הקונקרטי (צד ימין)

$\underline{Down - Casting}$:

"שנמוך אובייקט" – למשל להפוך את $Animal$ ל-Cow.

חייב להיות $Explicit$ - הקומפיילר מוודא שזה בוצע במכוון.

(הפעולה הזו תגרום לנו לאבד פונקציונליות מסוימת ולכן היא לא מומלצת !)

דוגמה 2 (דוגמה לשימוש לא נכון):

$Animal\ animal = new\ Cow();$

$Cow\ c = animal;$

הפקודה תגרור שגיאת קומפילציה, כי לא ציינו במפורש שאנחנו מעוניינים ב-$Down - Casting$.

דוגמה 1 (דוגמה לשימוש נכון):

$Animal\ animal = new\ Cow();$

$Cow\ c = (Cow)\ animal;$

$\underline{Implicit - Casting}$:

ביצוע $Cating$ באופן בלתי מפורש. (הדוגמאות שממוספרות עם 1 למעלה)

$\underline{Explicit - Casting}$:

ביצוע $Casting$ באופן מפורש. (הדוגמאות שממוספרות עם 2 למעלה)

לרוב לא תהיה סיבה לממש כך מכיוון שהקומפיילר יודע לזהות לבד כאשר מדובר ב-$Casting$.

חשוב לזכור:

לרוב נמנע מ-$Casting$ כי הוא יכול להיכשל בזמן ריצה.

אנחנו לומדים את זה כדי לדעת במה לא להשתמש..

Instanceof:

הסינטקס $instanceof$ עוזר לנו לבדוק האם אובייקט מסוים הוא מופע של אובייקט אחר. למשל:

$Animal\ animal = new\ Cow();$

נוכל להשתמש ב:

$if(animal\ instanceof\ cow)$ שיחזיר $True$,

כדי לבדוק האם $animal$ הוא מופע של Cow.

השימוש בסינטקס הזה לא מומלץ

(לרוב מוגדר כפתרון לא טוב, יוצר קוד מועד לבאגים ועוד)

Design Patterns

$delegation$ – מבנה עיצובי המשלב את היתרונות של $Composition$ ושל ירושה.

(ראינו לפני זה שלרוב נעדיף להשתמש ב-$Composition$ כדי לבצע שימוש חוזר בקוד)

נניח שאנחנו מעוניינים להשתמש בפונקציה foo של מחלקה A, במחלקה B שלנו.

לפי $Composition$, נוסיף שדה שמכיל אובייקט מ-A ואז נייצר שיטה חדשה בשם foo שכל מה שהיא תעשה זה לקרוא לשיטה foo שב-A. דוגמה:

```
public class B {
    private A a;
    public B(A a){
        this.a = a;
    }

    puclib void foo() {
        a.foo();
    }
}
```

וככה נוכל להשתמש ב-$B.foo()$ שתקרא ל-$A.foo()$ כמו שרצינו.

<u>יתרונות ל-$delegation$:</u>

- מאפשר להחליף אובייקט מסוג A באובייקט אחר בזמן ריצה לפי תנאי הסביבה.
- רוצים ש-B יירש ממחלקה אחרת (ואפשר לרשת רק ממחלקה אחת)
- תבנית כללית שטובה להרבה מקרים והקשרים שונים.

Facade:

$Design\ Pattern$ של מבנה (ארכיטקטורה)

הבעיה ש-$Facade$ בא לפתור:

רלוונטי למערכת גדולה עם הרבה מחלקות, כאשר יש API מורכב שקשה לעבוד איתו.

הרעיון של $Facade$ הוא לייצר תת-מחלקה חדשה, שתכיל שיטות פשוטות יותר המשתמשות בשיטות הקיימות ב-API המורכב, וכך תקל על השימוש במחלקות. (קל יותר להבין דרך הדוגמה הבאה)

<u>דוגמה</u>:

נניח שיש לנו מחלקות המייצגות מטבח, כלי מטבח, מצרכים וכו'.

כדי לייצר פסטה למשל, נצטרך להשתמש במגוון שיטות מה-API, ולא תמיד זה יהיה פשוט.

לכן נוכל לייצר מחלקת $Facade$ חדשה בשם $Chef$ שיהיו בה מחלקות שמכינות עבורנו מנות מסוימות למשל $MakePasta$, כך שאם הלקוח מעוניין להכין פסטה, הוא לא צריך את כל המחלקות הנ"ל אלא רק את ה-API הפשוט יותר של ה-$Facade$.

יתרונות:

- הלקוחות מתמודדים עם API מצומצם יותר.
- שינויים במחלקות הגדולות לא יעניינו את הלקוחות אלא רק את ה-$Facade$, (אם היא ממומשת בצורה נכונה היא תהיה עמידה לשינויים)
- אנחנו לא חוסמים לקוחות שמעוניינים בפונקציונליות גדולה יותר להשתמש במחלקות הגדולות.

חשוב לזכור: $Facade$ לא מוסיף פונקציונליות חדשה למחלקות אלא רק מפשט אותן.

Collections

מה זה $Collection$?

$Collection$ הוא אובייקט תכנותי שמחזיק הרבה אובייקטים אחרים.

למשל מערך, רשימה, רשימה מקושרת וכו'. נקרא גם $Data\ Structres$ או $Containers$.

משמש לאחסון מידע ושליפה שלו, ביצוע מניפולציות על המידע כמו למשל מיון, חיפוש ועוד'.

סביבת העבודה $Collections$:

ב-$Java$ קיימת סביבת עבודה ($FrameWork$) שימושית מאוד בשם $Collections$.

כדי להשתמש בה, נייבא את סביבת העבודה ע"י $import\ java.util.*$.

סביבת העבודה מכילה ממשקים ($Interfaces$) של מבני נתונים ללא מימוש

למשל $Set,\ List$ וכו', ובנוסף מכילה מימושים שונים של אותם הממשקים.

למשל $HashSet, TreeSet, LinkedList, ArrayList$ ועוד המון..

כל מבנה נתונים כזה מכיל שיטות שונות כגון חיפוש, מיון, ערבוב וכו' בזמני ריצה יעילים ככל האפשר.

$Intro\ to\ generics$:

מחלקה גנרית זו מחלקה שיכולה לקבל יותר מסוג אחד של פרמטרים כקלט.

למשל מחלקה בשם $List < E >$ זו מחלקה גנרית שמקבלת אובייקט מסוג שעדיין לא ידוע, וכרגע מסומן ב-$< E >$. גם השיטות של המחלקה יכולות להשתמש באותו משתנה למשל:

$E\ List.get(int\ index)$

אפשר לחשוב על E כ-$placeholder$, כלומר עד שנדע איזה סוג משתנה המתכנת מעוניין לשלוח, נשתמש בסינטקס $< E >$ בכל מקום, וברגע שהמחלקה תקבל סוג ספציפי של משתנה, $< E >$ "יהפוך" להיות הסוג הרלוונטי.

דוגמה ליצירת רשימה מקושרת המכילה מחרוזות:

$LinkedList < String > stringList = new\ LinkedList < String > ();$

הגדרנו רשימה מקושרת חדשה שתכיל אך ורק מחרוזות (כי קבענו שהסוג יהיה $String$)

ואם ננסה להוסיף לרשימה Int (למשל), נקבל שגיאת קומפילציה.

מעבר בלולאה על מבנה הנתונים:

כל מבנה נתונים של $Collection$ הוא $Iterable$ ולכן נוכל להשתמש בלולאת $foreach$ בצורה הבאה:

```
for(String str: strList)
        System.out.println(str);
```

(כל מבנה נתונים מממש שיטה שנקראת $iterator$ שמחזירה איטרטור,

והלולאה יודעת לבקש מהאובייקט איטרטור כזה בעצמה)

Collection Interfaces

$Collections$ מגדיר 7 ממשקים בסיסיים, הממשק הכי בולט הוא $Collection < E >$.

הממשקים $Queue < E >$, $List < E >$, $Set < E >$ יורשים מ-$Collection < E >$,

ו- $SortedSet < E >$ יורש מ-$Set < E >$.

בנוסף, קיימים עוד 2 ממשקים $Map < K, V >$ ו-$SortedMap < K, V >$ (עם 2 פרמטרים גנריים)

$List < E >$ (רשימה)

מבנה נתונים ששומר על סדר. (במובן שמוגדר מה האיבר הראשון, השני השלישי ..)

מכיוון שניתן לגשת לפי אינדקסים, הממשק מציע שיטות כמו $get(\quad), set(\quad), indexOf(\quad)$

רשימה יכולה להכיל את אותו איבר יותר מפעם אחת.

$Queue < E >$ (תור)

מבנה נתונים עם סדר המאפשר גישה רק לראש התור. הממשק מספק את השיטות:

- $Push()$ – להכניס איבר חדש לתור.
- $Peek()$ – מחזיר לנו את ראש התור.
- $Pop()$ – מאפשר להעיף את ראש התור, והבא אחריו יהפוך להיות ראש התור החדש.

לרוב, תור ממומש ע"י $FIFO$ $(first - in - first - out)$ כלומר הראשון שנכנס יהיה הראשון לצאת,

כמו למשל תור בקולנוע כדי לקנות כרטיס. הראשון שהגיע יהיה הראשון לקנות.

ניתן לממש גם לפי $priority\ queues$ (תור עדיפויות) כמו למשל בבתי חולים, שם חומרת הפציעה מחליטה על הסדר של התור.

$Set < E >$ (רשימה)

מבנה נתונים המכיל רצף של איברים ללא סדר ולא יכול להכיל איברים כפולים.

$SortedSet < E >$ (רשימה עם סדר)

מבנה נתונים המכיל רצף של איברים עם סדר ולא יכול להכיל איברים כפולים.

$Maps < K, V >$ (מיפוי בין מפתחות לערכים)

$K = keys, V = values$. מבנה נתונים שממפה מפתחות לערכים.

למשל מיפוי בין מספר סטודנט לשם הסטודנט וכו'.

המפתחות חייבים להיות שונים זה מזה, כמו ב-Set, אך אין הגבלה על כפילות של הערכים.

למשל יכולים להיות 2 מספרי סטודנט $(keys)$ שונים זה מזה שממופים ל-2 סטודנטים בעלי אותו שם.

$SortedMap < K, V >$ (מיפוי בין מפתחות לערכים עם סדר)

פועל באותו אופן כמו $Maps$ (עם כל ההגבלות) ואנלוגי ל-$SortedSet$.

למשל מילון שמכיל מילה ואת התרגום שלה יהיה מסודר לפי סדר א-ת.

Constructors

כל מבנה נתונים ב-$Collections$ ממש 2 בנאים:

- בנאי $void$ כלומר מבנה נתונים ריק.
- בנאי העתקה ($Copy\ Contructor$) עם פרמטר בודד מסוג $Collection$, שמחזיר אובייקט מסוג $Collection$ עם אותם ערכים שיש בפרמטר המקורי. מאפשר ליצור העתק של כל מבנה נתונים, למבנה הנתונים הספציפי הרצוי.

- חשוב להבין כי לא ניתן לחייב מחלקות יורשות, לייצר בנאים מסוימים, לכן מדובר במוסכמה. בסביבת העבודה $Collections$, כל המחלקות ממשות את הבנאים הנ"ל.

Collection Implentations

$ArrayList < E >$ (מערך בגודל משתנה)

מממש את הממשק $List < E >$. מבנה נתונים דומה למערך אך עם גודל משתנה.

- מגדיר פעולות של $get(\quad), set(\quad)$ בזמן קבוע
- פעולות כמו $contains(\quad), indexOf(\quad), remove(\quad)$ ב- $O(n)$
- הוספה למערך מתבצעת ע"י $add(\quad)$ ב- $O(1)$ ברוב המקרים (במקרים מסוימים $O(n)$)

$LinkedList < E >$ (רשימה מקושרת)

מממש את הממשק $List < E >$. מבנה נתונים של רשימה מקושרת.

- הוספה לרשימה מתבצעת ע"י $add(\quad)$ ב- $O(1)$
- $get(\quad), set(\quad), contains(\quad), indexOf(\quad), remove(\quad)$ ב-$O(n)$
- דורש פחות מקום בזיכרון מאשר $ArrayList$

$TreeSet < E >$ (סט הממומש ע"י עץ)

מממש את הממשק $Set < E >$ בעזרת עץ.

- לאיברים יש סדר.
- $add(\quad), remove(\quad), contains(\quad)$ ב-$O(\log n)$

$HashSet < E >$ (סט הממומש ע"י טבלת גיבוב)

מממש את הממשק $Set < E >$ ע"י טבלת גיבוב של $Java$.

- אין סדר לאיברים.
- $add(\quad), remove(\quad), contains(\quad)$ ב-$O(1)$ בממוצע !

$HashMap < K, V >$ (מיפוי בעזרת טבלת גיבוב)

$TreeMap < K, V >$ (מיפוי בעזרת עץ)

שניהם ממומשים בדיוק כמו $HashSet$ ו-$TreeSet$, אך לכל מפתח יש ערך.

דוגמה ליצירת סט המכיל מחרוזות:

$Set < String > str = new\ HashSet < String > ();$

כלומר הגדרנו סט חדש הממומש בעזרת $HashSet$, שיכיל אך ורק מחרוזות.

גם כאן, אם ננסה להוסיף לסט Int (למשל), נקבל שגיאת קומפילציה.

נשים לב כי במקרה זה הגדרנו את מבנה הנתונים $HashSet$ בתוך משתנה מסוג Set.

שיטות נפוצות לניהול המבנה - זמן הריצה שלהן (בממוצע) הוא $O(1)$:

- $add(E\ element)$
- $Contains(Object\ O)$
- $remove(Object\ O)$
- לעיתים רחוקות לא מדובר ב-$O(1)$.

אופרטורים של Set:

- איחוד $Union$ - $oneToFour.addAll(threeToFive)$ – יחזיר $\{1,2,3,4,5\}$
- חיסור $difference$ - $oneToFour.removeAll(threeToFive)$ – יחזיר $\{1,2\}$
- חיתוך $intersection$ - $oneToFour.retainAll(threeToFive)$ – יחזיר $\{3,4\}$

TreeSet

- סט עם סדר.
- מכיל סוגי משתנים/אובייקטים שיש ביניהם סדר. $(Integers, Strings\ \ldots)$

דוגמה ליצירת $TreeSet$ המכיל מחרוזות:

$NavigableSet < String > str = new\ TreeSet < String > ();$

(אותו הסבר כמו מקודם)

שיטות נפוצות לניהול המבנה - זמן הריצה שלהן הוא $O(\log n)$:

- $add(E\ element)$
- $Contains(Object\ O)$
- $remove(Object\ O)$

מכיוון שלמימוש הזה של סט יש סדר, יש לו את השיטות $first(), last()$

ואת השיטה $descendingIterator()$ – איטרטור מהגבוה לנמוך.

Comparable

כדי שנוכל להשתמש בפעולות כמו $Sort$, לאיברים שנמצאים בתוך מבנה הנתונים חייב להיות סדר.
למשל ש-$1 > 0$ וכו'. באובייקטים שונים כמו למשל $ComplexNumber$, אנחנו נצטרך להגדיר מי גדול ממי על ידי מימוש האינטרפייס $Comparable$ ומימוש השיטה $compareTo$.
המימוש אמור להראות כך: (בפסאודו-קוד)

```
compareTo(Other)
        -1  if this is smaller
        0   if this.equals(Other)
        1   if this is bigger
```

האינטרפייס הוא גנרי. לא נגדיר את ה-$Interface$ לקבל משתנה מסוג $Comparable$,
כי אז השיטה יכולה לקבל כל אובייקט שמממש את $Comparable$,
אבל אין שום הגיון להשוות בין מחרוזת לבין int. לכן האינטרפייס מוגדר כך:

```
public Interface Comparable < T > {
        int compareTo(T other);
}
```

ומימוש מחלקה שתהיה $Comparable$ יראה כך:

```
public class ComplexNumber implements Comparable < ComplexNumber > {
        public int compareTo(ComplexNumber other) {
                return Double.compare(this.getReal(), other.getReal());
        }
}
```

Iterators

איטרטור הוא אובייקט שמכיל 2 שיטות עיקריות:

$hasNext()$ – מחזיר $True$ אם יש עוד איברים במערך, $False$ אחרת.

$next()$ – מחזיר את הערך הבא באיטרטור.

מימשנו איטרטור של $Integers$ למשל ב-$ex4$, על ידי יצירת מחלקה מקוננת באופן הבא:

```
public Iterator < Integer >  iterator() {
        // יוצרים מחלקה "מקוננת" חדשה בתוך השיטה, שכל אובייקט שלה הוא איטרטור
    class SomeName implements Iterator < Integer > {
        // כאן נוסיף משתנה שישמור כל פעם את האיבר הבא שצריך להחזיר
        @Override
        public boolean hasNext() {
            // נחזיר ערך בוליאני האם קיים איבר נוסף באיטרטור או לא
        }
        @Override
        public Integer next() {
            // נחזיר את האיבר הנוכחי ונעדכן את המשתנה שיחזיק את האיבר הבא
        }
    }
        // נחזיר אובייקט חדש של המחלקה הזאת, שהוא בעצם האיטרטור
    return new SomeName();
}
```

<u>שימוש באיטרטור עם לולאת $While$:</u>

```
List < String > myList = ⋯;
Iterator < String > myIterator = myList.iterator();
while(myIterator.hasNext()) {
    String next = myIterator.next();
    System.out.println(next);
}
```

<u>שימוש באיטרטור עם לולאת $Foreach$:</u>

```
for(String str : myList){
    System.out.println(str);
}
```

Exceptions

<u>סוגי שגיאות:</u>

$Compiliation\ Error$ – שגיאות קומפילציה, כלומר שגיאות שמזוהות עוד בשלב כתיבת הקוד, שהקומפיילר מזהה ומתריע בפניהן.

$Runtime\ Error$ – שגיאות זמן ריצה, שגיאות שהקומפיילר לא מזהה. כמו למשל קלט לא נכון מהמשתמש, ניסיון גישה לקובץ לא קיים וכו'.

<u>הערה</u>: מתכנת טוב יטפל בשגיאות כשאפשר, ויציין בדוקומנטציה גם את המקרים בהם אי אפשר.

<u>הערה 2</u>: לכל מחלקה שמייצגת $Exception$ נוסיף:

$$private\ static\ final\ long\ serialVersionUID = 1L;$$

(בהמשך הסיכום יש הסבר מורחב על המשתנה הזה, אבל מומלץ להתקדם לפי הסדר כדי להבין)

<u>הערה 3</u>: נזרוק שגיאה במקרים בהם אין משמעות לערך ההחזרה במקרה של שגיאה.

(למשל במקרים בהם בלית ברירה נצטרך להחזיר $null$, נשקול שימוש במכניזם של $Exception$)

$Exceptions$ הם אובייקטים לכל דבר. לכן כשנבצע $throw$ (כלומר נזרוק אותם), נזכור שעלינו לזרוק אובייקט שגיאה חדש ולא רק את השגיאה. כלומר $throw\ \mathbf{new}\ NoSuchElementException()$.

מחלקה שמממשת שגיאה, תכיל בנאי ריק ובנאי שמקבל מחרוזת, למקרה בו נרצה להוסיף הודעה אינפורמטיבית בנוסף לשגיאה. למשל: $throw\ \mathbf{new}\ NoSuchElementException("msg")$

נזכור תמיד לתעד בדוקומנטציה שהשיטה זורקת שגיאה בעזרת המילה השמורה $@throws$:

```
/**
* @throws NoSuchElementException if the user calls next when hasNext returns false
*/
```

<u>$try, catch$:</u>

נשתמש ב-$try, catch$ כאשר נרצה לבצע פעולה מסוימת שיכולה לא להצליח. כלומר במקרה בו אולי נצטרך לזרוק שגיאה. למשל עבור רשימה, אם נתקלנו בשגיאה, המחלקה תזרוק $ListException$.

נתפוס אותו ב-$Catch$ בצורה הבאה:

```
try {
        int element = myList.get(0);
catch(ListException e){
        // handle error
}
```

נשים לב שייצרנו אובייקט $ListException$ בשם e, כך שנוכל להשתמש בו בתוך הסקופ של $Catch$.

(למשל לבדוק האם קיים מידע נוסף לגבי השגיאה באובייקט עצמו וכו')

Packages

$Package$ – חבילה, דרך לאגד מחלקות וממשקים קשורים ביחד.

למשל כשאנחנו משתמשים בשיטות סטטיות של $Math$,

אנחנו בעצם משתמשים ב-$Package$ של מחלקות מתמטיות שונות, שאוגדו תחת חבילה אחת.

כך נוכל בעתיד לשתף את החבילה שלנו או לבצע בה שימוש חוזר בפרויקטים אחרים.

אם ייצרנו $Package$ בשם $pack1$,

נזכור שכל קובץ צריך להכיל בתחילת הקובץ את הפקודה: $package\ pack1;$

מחלקות $Exceptions$ שבנינו למחלקות מתוך ה-$package$ יכללו גם הן באותו $package$.

<u>הרשאת $package - Private$:</u>

עד לא מזמן היינו רגילים לתת לכל שיטה או מחלקה $modifier$ $(pulic,\ private,\ protected)$

הרשאת $package - private$ או הרשאת $default$ (אותו דבר) מציינת שכל מחלקה/שיטה וכו'

שלא מציינת שום $modifier$ היא בעצם $package - private$ כלומר ניתנת לגישה אך ורק על ידי

הקבצים שבאותו $package$.

למשל אם במקום $public\ class\ someName$ נרשום $class\ someName$,

המחלקה $someName$ תהיה נגישה לכל המחלקות ב-$package$,

אך שום מחלקה מבחוץ לא יכולה להשתמש בה.

סיכום מעולה להבדל בין ה-$modifiers$ השונים (כאשר $no\ modifier$ הוא בעצם ה-$default$):

```
            | Class | Package | Subclass | Subclass | World
            |       |         |(same pkg)|(diff pkg)|
------------+-------+---------+----------+----------+--------
public      |   +   |    +    |    +     |    +     |   +
------------+-------+---------+----------+----------+--------
protected   |   +   |    +    |    +     |    +     |
------------+-------+---------+----------+----------+--------
no modifier |   +   |    +    |    +     |          |
------------+-------+---------+----------+----------+--------
private     |   +   |         |          |          |

+ : accessible
blank : not accessible
```

Nested Classes

מחלקה "מקוננת" – כלומר יצירת מחלקה בתוך מחלקה אחרת. למשל:

```
class A {
        \\ Some methods
        class B {
                \\ Some methods
        }
        \\ Some methods
}
```

<u>סיבות עיקריות לשימוש במחלקות מקוננות:</u>

- קיבוץ לוגי של מחלקות. למשל כאשר המחלקה B רלוונטית אך ורק למחלקה A.
- הגדלת האינקפסולציה. המחלקה B יכולה לגשת למשתנים של מחלקה A, גם אם הם $private$ (כי הם באותו $scope$). אם B אכן משתמשת במשתנים של A והיינו מגדירים אותה כמחלקה נפרדת, היינו צריכים להפוך את המשתנים האלה ל-$public$.
  לכן מחלקות מקוננות מחזקות / שומרות על האינקספולציה.
- נוכל להגדיר את B להיות מחלקה פרטית ובכך היא לא תהיה זמינה למחלקות אחרות.

<u>כללי אצבע:</u>

- מחלקות $Exceptions$ **לא** יהיו מחלקות מקוננות אלא מחלקות רגילות.
- מחלקות מקוננות צריכות להיות קטנות. (אחרת הקוד מסורבל מדי)
- להבדיל ממחלקות רגילות, שיכולות להיות רק $public$ או $package - private$,
  מחלקות מקוננות יכולות להיות $private,\ public,\ protected,\ package - private$
- לרוב נשאף להגדיר אותן כ-$private$. (אם יש הגיון ל-$modifier$ אחר, אולי המחלקה לא צריכה להיות מחלקה מקוננת אלא מחלקה רגילה)

יש 2 סוגי מחלקות מקוננות עיקריות:

- $Static\ Nested\ Class$
- $Inner\ Class$
- $Local\ Class$
- $Anonymous\ Class$

נרחיב בעמודים הבאים.

Static Nested Classes

נשתמש במחלקה מקוננת סטטית אם אין קשר בין ה-*instance* של המחלקה העוטפת למחלקה הפנימית.

<u>נראה בעזרת דוגמה:</u>

```
public class EnclosingClass {

    private int dataMember = 7;

    public void createAndIncrease() {
        NestedClassin = new NestedClass();
        in.nestedDataMember++; // a private field of the nested class
    }

    private static class NestedClass{
        private int nestedDataMember = 8;
        private void nestedCreateAndIncrease() {
            EnclosingClassen = new EnclosingClass() ;
            en.dataMember++ ;// a private field of the enclosing class
        }
    }
}
```

<u>הסבר על הדוגמה:</u>

- במקרה זה בו המחלקה הפנימית היא סטטית, כל מחלקה יכולה לגשת למשתנה *private* של מחלקה אחרת **רק** אם היא מייצרת *instance* שלו. נשים לב בדוגמה שכל מחלקה מייצרת *instance* של המחלקה השנייה כדי לגשת למשתנים שלה.

Inner Class

נשתמש במחלקה מקוננת פנימית (שאיננה סטטית) אם יש קשר בין ה-$instance$ של המחלקה העוטפת למחלקה הפנימית.

- יכולה לגשת ל-$data\ members$ של המחלקה העוטפת.
- לא ניתן להגדיר בה משתנים סטטיים, מכיוון שהיא מקושרת למחלקה שאפשר ליצור ממנה $instance$ ולכן אין הגיון לכך.
- לא יכול לחיות בלי $instance$ של המחלקה העוטפת.

<u>נראה בעזרת דוגמה:</u>

```
public class EnclosingClass {
    private int dataMember = 7;
    public void createAndIncrease() {
        MemberClassmem = new MemberClass();
        mem.memberClassField ++;
    }

    private class MemberClass {
        private int memberClassField = 8;
        private void memberClassIncrease() {
            dataMember ++ ;// a privatefield of the enclosing class
        }
    }
}
```

<u>הסבר על הדוגמה (והבדלים מהדוגמה הקודמת):</u>

- במקרה זה המחלקה העוטפת כן צריכה לייצר $instance$ של המחלקה הפנימית כדי להשתמש בה.
- ההבדל מהדוגמה הקודמת הוא שהמחלקה הפנימית לא צריכה לייצר $instance$ של המחלקה החיצונית, אלא היא יכולה לגשת לכל המשתנים (גם ל-$private$) בלי בעיה.

Local Classes

מחלקה שמוגדרת בתוך שיטה. הדוגמה הנפוצה ביותר למקרה זה היא מימוש $Iterator$.
בעמוד הסבר על $Iterators$ הראנו דוגמה למימוש איטרטור. ביצענו את זה על ידי מימוש
$Local\ Class$ בתוך שיטה בשם $Iterator$. נשתמש במחלקות לוקאליות כאשר אין סיבה להגדיר את
המחלקה מחוץ ל-$Scope$ של השיטה.

מחלקה לוקאלית לא יכולה להיות $public,\ private,\ protected$ או $static$.
מגדירים אותה ללא $modifier$.

Anonymous Class

למשל עבור ה-$interface$ הבא:

```
interface FilenameFilter {
        boolean accept(File dir, String s);
}
```

נוכל לייצר מחלקה אנונימית באופן הבא:

```
String[] filelist = dir.list(
        new FilenameFilter() {
                public boolean accept(File dir, String s) {
                        returns s.endsWith(".java");
                }
        }
);
```

כלומר ייצרנו מערך של מחרוזות בשם $filelist$. נשים לב ש-new במקרה זה לא מציין יצירת
$instance$ של ה-$interface$ (הגיוני כי זה $interface$), אלא יצירת מחלקה אנונימית חדשה
שמממשת את ה-$interface$ הזה.

Modularity

הרעיון של מודולריות הוא לקחת את התוכנה שלנו ולפרק אותה ליחידות קטנות ובלתי תלויות. היחידות האלה נקראות מודולים.

יתרונות לתכנות מודולרי:

- יותר קל לתחזוק (דיבוג, הרחבה וכו')
- מפרק בעיות גדולות לבעיות קטנות מה שמקל על המימוש ועל הדיבוג.
- מאפשר חלוקה של בניית התוכנית לצוותים שונים / מתכנתים שונים.

ארבעה עקרונות עיקריים למודולריות:

- <u>$Decomposability$ – פריקות:</u>

  נפרק בעיה לתתי בעיות קטנות יותר.

  הנקודה החשובה ברעיון הזה הוא חיבור פשוט בין הבעיות.

  לא נוכל להשתמש בפריקות למשל כאשר אתחול התוכנה דורש "נגיעה" בכל אחד מהמודולים, מכיוון שמאוד קשה לפרק את הבעיה לחלקים במקרה הזה.

- <u>$Composability$ – הרכבה:</u>

  להבדיל מהגישה הקודמת, הגישה הזו דוגלת בשימוש של יחידות קטנות קיימות והרכבה שלהן לתוכנית גדולה יותר. המודולים יהיו אוטונומיים (כלומר לא תלויים אחד בשני) בנוסף, הגישה הזו קשורה ישירות למחזור קוד, כי כל מודול הוא אוטונומי וניתן לשימוש במחלקות אחרות ללא קשר לשאר המודולים. (ואנחנו אוהבים מחזור קוד 😉)

- <u>$Understandability$ – מובנות:</u>

  קורא אנושי שיסתכל על המודולים שלנו, יכול להבין את כל המודולים בנפרד בלי להבין גם את שאר המודולים. כלומר כל מודול הוא אוטונומי מספיק כך שאין צורך להסתכל על מודולים נוספים כדי להבין אותו. חשוב להבין שלא מדובר בקריאות אלא במובנות, כלומר הבנה של ה-$Design$, מה בדיוק עושה המודול ואיך הוא משתלב בתוכניות אחרות.

- <u>$Modular\ Continuity$ – רציפות:</u>

  אם יש שינוי בדרישות ואנחנו צריכים לשנות מודול מסוים, השינוי ישפיע על המודול הזה בלבד (ואולי על כמה נוספים) אבל לא יגרום לנו לעבור על כל המודולים מחדש.

הערה: פריקות והרכבה אומנם נראים מנוגדים, אך ניתן (אחרי ניסיון) לשלב ביניהם בהצלחה.

עקרונות נוספים למודולריות:

- $Open - Close\ Principle$ – עקרון פתוח-סגור

עקרון לפיו רכיבי תוכנה צריכים להיות סגורים לשינוי אבל פתוחים להרחבה.
כלומר אם משנים את הדרישות, ניצור קוד חדש ובכך נרחיב את המודול.
קוד קיים שכבר עבר דיבוג, נמצא אצל לקוחות ועובד תקין, אנחנו לא רוצים לשנות אותו.

- $Single - Choice\ Principle$ – עקרון בחירה יחידה

נתאר את העקרון בעזרת דוגמה. נניח שיש לנו רשימת צורות (ריבוע, עיגול וכו') שאנחנו רוצים לצייר.
בנוסף יש לנו שיטה שמאפשרת למשתמש לרשום בעצמו צורות ולהדפיס אותן. ב-2 המקרים נצטרך סוג של "מילון צורות", כלומר אם המחרוזת היא "ריבוע" אז תדפיס ריבוע וכן הלאה.
במקרה זה, נשאף שהמילון יהיה במקום אחד ויחיד וכל מודול ישתמש בו.
כך אם נצטרך למחוק צורה או להוסיף צורה, נשנה את הקוד ברשימה הספציפית הזאת,
ולא נצטרך לגעת במודולים האחרים.

Factory Design Pattern

אובייקט שמטרתו לייצר אובייקטים אחרים (ממש כמו מפעל)
הרעיון הוא לייצר הפרדה בין הקוד שמייצר את האובייקטים לבין הקוד שמשתמש בהם (מודולריות)
ה-$Factory$ יכול להיות ממומש כמחלקה אבסטרקטית או כשיטה (למשל שיטת $loadAll()$ כלשהי)
ולרוב נגדיר את $Factory$ להיות מחלקת $Singleton$ (כי אין סיבה לייצר 2 מופעים מהמחלקה)

Singleton Design Pattern

במקרים בהם נרצה שיהיה לנו אובייקט אחד ויחיד של מחלקה מסוימת.
למשל ה-$Task\ Manager$ של $Windows$ הוא אחד ויחיד, לא נרצה שיהיו 2 כאלה כדי שלא יתנגשו.
נרצה למנוע יצירת $instances$ חדשים, ונרצה שתהיה גישה נוחה אליו.

נגדיר למשל מחלקה בשם $Singleton$ המקיימת את 3 התנאים הבאים:

1. יש לה משתנה **סטטי** ו-$private$ של האובייקט הזה במחלקה,
2. הבנאי היחיד של המחלקה הזאת יהיה גם הוא $private$.
3. יש לה מתודה **סטטית** בשם $instance()$ שתחזיר את המשתנה הסטטי.

```
public class Singleton {
    private static Singleton single = newSingleton();
    private Singleton () { ... }
    public static Singleton instance() {
        return single;
    }
}
```

<u>נשים לב:</u>

- לא ניתן לרשת את המחלקה הזאת בגלל שהבנאי שלה מוגדר כ-$private$.
- היתרון של $Singleton$ לעומת מחלקה סטטית, הוא שמחלקת $Singleton$ היא מחלקה רגילה ולכן יכולה לממש $interfaces$, ניתן לבצע $up-casting$ וכו'.

Strategy Design Pattern

$Design\ Pattern$ התנהגותי. עקרון זה שואף להפריד בין החלק התכנותי לחלק ההתנהגותי של התוכנית, כך שנוכל "לשנות" את ההתנהגות של התוכנית שלנו תוך כדי ריצה.
למשל מחלקה שממיינת מערך. המחלקה מקבלת מערך לא ממוין ומחזירה מערך ממוין,
אך אופן המימוש יכול להשתנות. למשל שימוש ב-$Bubble\ Sort$ ותוך כדי ריצה להבין שעדיף לנו להשתמש ב-$QuickSort$ וכו'.

נוכל לבצע זאת למשל על ידי הגדרת API מסוים ($abstract\ \&\ interface$),
וכל התנהגות שונה תוכל לממש את ה-API הזה.

Streams

קבלה ושליחה של מידע. יש הרבה מקורות מידע שיכולים להתממשק עם התוכנית שלנו, כמו למשל מדפסות, אינטרנט, תוכנות אחרות, דיסק קשיח וכו'. נשים לב שכדי לממש את קישורים בין כל הגורמים הנ"ל, נצטרך לממש לכל גורם: (ה-$stream$ הוא "צינור מידע")

- שליחת מידע $(put\ information\ into\ stream)$
- קבלת מידע $(get\ information\ from\ stream)$

<u>$Java\ Stram\ Library$</u>

ספריית $Streams$ של $Java$ מאפשרת הפשטה נוחה לעבודה של כל תוכנה עם מגוון מקורות אחרים כפי שהזכרנו לעיל. הספרייה מחביאה את רוב הפרטים ממי שמשתמש ב-$Stream$, היא מתייחסת לכל מקור מידע כ"קופסה שחורה", כלומר אנחנו צריכים לדאוג רק לקבל מידע ולשלוח מידע.

<u>אנלוגיה:</u>

מצד אחד, לא מעניין אותנו מאיפה מגיעים המים או באיזה צינורות $(pipes)$ אנחנו משתמשים, העיקר שהמים יוצאים מהברז. מצד שני, המים נשטפים ולא מעניין אותנו לאן או איך.

<u>API בעבודה עם $Streams$:</u>

$Create$ – ליצור התקשרות עם גורם אחר.

$Write$ – שליחת/כתיבת מידע לגורם האחר.

$Read$ – קריאת מידע מהגורם האחר.

$Delete$ – היכולת לסגור / למחוק את ההתקשרות עם הגורם האחר.

<u>קיימים 2 מקורות מידע עיקריים:</u>

מידע טקסטואלי – מקורות מידע המכילים טקסט. (למשל קבצי $txt, word, py, java$ וכו')

מידע בינארי – מקורות מידע המכילים רצפים של בייטים $(Bytes)$ (למשל קבצי $jpg, mp3, zip$ וכו')

<u>$Data\ Encoding$ – קידוד מידע:</u>

בעבודה עם $Streams$ נצטרך להסכים על אופן הקידוד

(מבנה המידע, טקסטואלי או בינארי, מה כל רצף של אותיות או בייטים מציין וכו')

כל צד אחראי לציין את אופן קידוד המידע שלו. ("באיזה שפה הוא מדבר")

ב-$Java$ יש מחלקה $java.io.*$ שמשמשת לקבלה וקריאה של מידע.

היא מכילה 4 מחלקות אבסטרקטיות:

- $Reader$ ו-$Writer$ – לעבודה עם קבצים טקסטואליים.
- $InputStram$ ו-$OutputStream$ – לעבודה עם קבצים בינאריים.

מהמחלקות הנ"ל יורשות מחלקות נוספות, פירוט חלקי שלהן:

I/O Type	Streams
Memory	*CharArrayReader/Writer* *ByteArrayInput/OutputStream*
Files	*FileReader/Writer, FileInput/OutputStream*
Buffering	*BufferedReader/Writer, BufferedInput/OutputStream*
Data Conversion	*DataInput/OutputStream*
Object Serialization	*ObjectInput/OutputStream*
Filtering	*FilterReader/Writer, FilterInput/OutputStream*
Converting between bytes and characters	*InputStream/OutputReader*

דוגמה לשימוש:

```
Writer writer = new FileWriter("mail.txt");
writer.write('a');
writer.close();
```

במקרה זה כתבנו את האות a לתוך הקובץ $mail.txt$ ואז סגרנו את הקובץ.

דוגמה נוספת:

```
try(OutputStream output = new FileOutputStream(args[1]);
                InputStream input = new FileInputStream(args[0]); ) {
        int result;
        while((result = input.read()) != -1) {
        output.write(result);
        }
} catch(IOExceptionioe) {
        System.err.println("Couldn't copy file");
}
```

זו הדרך המומלצת לקריאת קבצים. נשים לב שאת פתיחת הקבצים ביצענו בתוך הסוגריים של try, עבור $Java$ 7 ומעלה זה אפשרי וכך אם נתקלנו בבעיה במהלך הריצה, הקבצים ייסגרו לבד. אחרת, אם השגיאה תקרה לפני שנבקש לסגור את הקבצים, הם יישארו פתוחים וזה מתכון לבעיות.

Decorator Design Pattern

$decorator$ – מקשט, הוא (עוד) $design\ pattern$, רלוונטי ל-$streams$ ולתחומים נוספים בתכנות.

<u>הבעיה</u>: יש לנו אוסף של $streams$ מסוגים שונים, ואנחנו רוצים להוסיף להם פונקציות נוספות.
(כיווץ, הצפנה וכו') אם נייצר לכל סוג כזה מחלקה שמרחיבה אותו, נסיים עם המון מחלקות והמון קוד משוכפל. לכן נרצה להרחיב את המחלקות בצורה טובה יותר.

<u>הפתרון</u>: נגדיר מחלקה A שמרחיבה את מחלקה B עם הפונקציונליות הנוספת כך:

- A תירש מ-B $(A\ extends\ B)$
- מבצעת $delegation$, כלומר למחלקה A יהיה שדה המכיל אובייקט של המחלקה B,
  והיא תעביר אליו בקשות (הסבר על $delegation$ יש בעמודים קודמים)

<u>$Buffer$ ו-$Compress$:</u>

2 דרכים נפוצות לייעול תוכנית שקוראת / כותבת לקבצים אחרים בעזרת $streams$ הן:

- כיווץ המידע לפני שליחתו.
  (למשל נרצה לכווץ מידע שנשלח מהפלאפון כדי לחסוך בשימוש חבילת הגלישה שלנו)
- שימוש ב-$buffer$ – קונטיינר של מידע בזיכרון הלוקאלי של התוכנית.
  נקרא כמות גדולה של מידע לתוכו, וכך נוכל לשלוף את המידע במהירות.
  באותו אופן נאגור בו מידע שנרצה לשלוח, ונשלח את כולו בפעם אחת.

$buffer$ חוסך בזמני הריצה מכיוון שעבור המחשב, לקרוא בייט אחד ולקרוא 1000 בייטים "עולה" בערך אותו דבר. אנלוגיה במקרה הזה היא למשל פינוי זבל של הבית. במקום ללכת לפח השכונתי בשביל כל פיסת נייר, נזרוק את הזבל בפח הביתי שקרוב אלינו, ונרוקן אותו פעם בכמה זמן לפח השכונתי. באופן הזה אנחנו חוסכים זמן והליכות מיותרות לפח השכונתי (עצלנים בקיצור)

<u>דוגמה לשימוש ב-$Buffer$:</u>

```
Reader inFile = new FileReader("my_file.txt");
Reader inBuffer = new BufferedReader(inFile);
Writer outFile = new FileWriter("my_file.txt");
Writer outBuffer = new BufferedWriter(outFile);
```

נשים לב שה-$decorator\ design\ pattern$ בא לידי ביטוי במקרה זה בכך "הגדרנו" מחלקה אחת $BufferedWriter$ שיודעת להתמודד עם כל סוגי הקבצים (הטקסטואליים).

<u>$Scanner$:</u>

מחלקה נפוצה ב-$Java$, משמשת לקריאת מידע טקסטואלי ומקבלת רכיב $InputStream$.
המחלקה מכילה פונקציות שימושיות לטובת אנליזה של טקסט, היא משתמשת ב-$delegation$,
יש לה $buffer$ מובנה, אך היא **לא** יורשת את $InputStream$ ולכן היא לא $decorating\ class$.

Enums

$Enum$ הוא אובייקט שמגדיר $type$ בצורה מהירה.

למשל: $public\ enum\ color\ \{WHITE,\ BLACK,\ RED,\ YELLOW,\ BLUE\};$

כל צבע כזה שהכנסנו הוא $enum$ מסוג $color$.

<u>דוגמה לשימוש:</u>

```
enum Season {WINTER, SPRING, SUMMER, FALL};

public Shirt chooseShirt(Season season){
        if (season.equals(Season.SUMMER)) {
                return new TShirt();
        } else
                return null;
}
```

כלומר וידאנו שה-$Season$ שקיבלנו הוא $SUMMER$.

$SUMMER$ הוא לא מחרוזת, מספר או משהו מוכר אחר אלא אובייקט $enum$ מסוג $Season$.

לכן לא נוכל לשלוח לשיטה $choostShirt$ מחרוזת או מספר, אך ורק $enum$ מסוג $Season$.

<u>שילוב של $Inner\ Classes$ עם $Enums$:</u>

```
public class ExampleClass {
        public enum Planet {
                MERCURY(3.333, 2.333),
                VENUS(4.869, 6,051),
                PLUTO(1.27, 1.137);

                private final double mass;
                private final double radius;

                Planet(double mass, double radius) {
                        this.mass = mass;
                        this.radius = radius;
                }

                public double mass() {
                        return mass;
                }
        }
}
```

כלומר הגדרנו 3 אובייקטים מסוג $planet\ enums$ שמכילים שדות,

כי הגדרנו שה-$enum$ במקרה זה הוא מחלקה עם בנאי שמקבל $mass$ ו-$radius$.

בנאי של $Enum$:

הבנאי חייב להיות $package - private$ או $private$.

אם ננסה להגדיר $public$ או $protected$ נקבל שגיאת קומפילציה.

הבנאי הדיפולטיבי מוגדר כ-$private$.

$values()$:

ל-$enum$ יש מתודה $values()$. (לכל $enum$ מכל סוג שנגדיר)

המתודה מחזירה את כל הערכים של ה-$enum$ הזה. למשל בדוגמה הנ"ל,

נקבל את: $MERCURY, VENUS, PLUTO$. ערך ההחזרה של השיטה הוא (סוג של) איטרטור.

כלומר נוכל לרוץ על הערכים בלולאת $foreach$ למשל: $for(Planet\ p : Planet.values())$

יתרונות לשימוש ב-$Enums$:

- $Type\ Safety$ – מוודא שהפרמטרים הנתונים הם באמת מסוג "עונה" או "כוכב" או "יום".
  (למשל עבור עונות, אם היינו מגדירים כל עונה כמחרוזת שמחזיקה את שם העונה,
  יכלו לשלוח לנו כפרמטר כל מחרוזת אחרת ואין לנו שום דבר להתמודד עם זה.
  במקרה של $Enums$, ניסיון לשלוח פרמטר עם $Type$ שונה יגרום לשגיאת קומפילציה
  (ולזה אנחנו שואפים, שגיאות קומפילציה מאשר שגיאות זמן ריצה)
- $Self\ explanatory$ – קוד ברור יותר.
- אפשר להשתמש ב-$valueOf$ שהופך סטרינגים לפעולות.

מתי לא להשתמש ב-$Enums$:

- אם הערכים לא ידועים כבר בזמן הקומפילציה. (למשל עבור מקרים של קלט מהמשתמש)

Generics

Intro to generics:

מחלקה גנרית זו מחלקה שיכולה לקבל יותר מסוג אחד של פרמטרים כקלט.

למשל מחלקה בשם $List < T >$ זו מחלקה גנרית שמקבלת אובייקט מסוג שעדיין לא ידוע, וכרגע מסומן ב-$< T >$. גם השיטות של המחלקה יכולות להשתמש באותו משתנה למשל:

$T\ List.get(int\ index)$. אפשר לחשוב על T כ-$placeholder$, כלומר עד שנדע איזה סוג משתנה המתכנת מעוניין לשלוח, נשתמש בסינטקס $< T >$ בכל מקום, וברגע שהמחלקה תקבל סוג ספציפי של משתנה, $< T >$ "יהפוך" להיות הסוג הרלוונטי.

הערה: בהסברים הבאים, נשתמש לרוב ב-$LinkedList$ ו-$List$ אך כמובן שאלה רק דוגמאות, וניתן לממש כל מחלקה באמצעות $Generics$.

מניעה לעומת טיפול:

המטרה שלנו בתור מתכנתים היא למנוע שגיאות מראש, ולא לטפל בהן אחרי שכבר קרו.

$Generics$ עוזר לנו בדיוק בשביל זה, כי הוא מבטיח לנו $Type - Safety$.

נסתכל על הדוגמה הבאה: $String\ s1 = new\ Integer(\quad);$

במקרה זה תתקבל שגיאת קומפילציה כי אנחנו מנסים להכניס אובייקט מסוג $Integer$ לתוך משתנה שיכול לקבל רק $String$. (כלומר $Compile - time\ type\ error$)

כעת נסתכל על הדוגמה הבאה:

$$Object\ o = new\ Integer(\quad);$$
$$String\ s2 = (String)o;$$

במקרה זה תתקבל שגיאת זמן ריצה, 2 הפעולות חוקיות ולכן הקומפיילר לא יזהה שמדובר בשגיאה. (אנחנו מנסים לעשות $Down - Casting$ לא חוקי) ובמקרה זה נקבל $run - time\ type\ error$.

זו הסיבה שתכנות $Type - Safety$ מבטיח לנו שגיאות בזמן קומפילציה במקום שגיאות זמן ריצה, ו-$generics$ זה מנגנון שמטרתו להבטיח $Type - Safety$.

Generics are Invariant:

נשים לב כי הפעולה הבאה:

$$LinkedList < Object > myObjList = new\ LinkedList < String > (\quad);$$

לא חוקית, מכיוון שלא ניתן לבצע $Up - Casting$ או $Down - Casting$ בין סוגים שונים של אובייקטים הנוצרים בעזרת $Generics$, אפילו ש-$String$ היא תת-מחלקה של $Object$.

(במקרה שלנו $LinkedList < Object >$ ו-$LinkedList < String >$)

נקודה חשובה: לאחר יצירת הרשימה עם $Object$ בצורה החוקית הבאה

$$LinkedList < Object > myObjList = new\ LinkedList < Object > (\quad);$$

נוכל להכניס מחרוזות לתוך הרשימה בלי בעיה בדיוק מהסיבה ש-$String$ היא תת-מחלקה של $Object$. (חשוב להבין את ההבדל בין ההערה לטענה שמעליה !)

Wildcards:

כדי להתמודד עם המגבלה הקודמת שהצגנו, נוכל להשתמש ב-*Wildcards*.

נוכל להשתמש למשל ב-$List <?>$ (סימן שאלה) כדי לציין שלא מעניין אותנו מה הסוג הספציפי של הרשימה. זאת אומרת אנחנו מוכנים לקבל כל סוג של אובייקט לתוך הרשימה,

מסיבה זו, ההנחה היחידה שאנחנו (המתכנתים) יכולים לעשות על הקלט בזמן בניית המחלקה (כלומר על סוג האובייקט שנכניס במקום סימן שאלה) היא שהוא יורש מ-*Object*.

למשל $List < Dog >, List < String >$ וכו'. מאותה סיבה, לא נוכל להכניס לרשימה אובייקט ממשי, אלא רק *null*, כי רק *null* יכול להיכנס **לכל** סוג של רשימה.

(מי שישתמש במחלקה שלנו, ויצור למשל $List < Dog >$, כן יוכל להכניס לתוכה אובייקטים מסוג *Dog*, הכוונה היא שבזמן בניית המחלקה $List <?>$ לא נוכל להכניס למשל אובייקט ברירת מחדל, כי אנחנו לא יודעים את סוג האובייקט עבורו תמומש הרשימה)

דוגמה למתודת *Factory* שמשתמשת ב-*Wildcards*:

אם נרצה להחזיר רשימה גנרית שאנחנו לא ידועים מראש מה יהיה הסוג שלה,

נוכל להשתמש במתודה שמחזירה $List <?>$:

```
private List <?> generateListFromUserInput(String str){
        switch(str){
                case "String": return new LinkedList < String > ();
                case "Integer": return new LinkedList < Integer > ();
                default: return new LinkedList < Object > ();
        }
}
```

למה לא להשתמש פשוט ב-$List < Object >$?

נסתכל למשל על השיטה:

```
void printList(List < Object > c) {
        for(Object e : c)
                System.out.println(e);
}
```

אמרנו שלא ניתן לבצע $Up - Casting$ או $Down - Casting$ בין סוגים שונים של אובייקטים הנוצרים בעזרת *Generics*, ולכן באותו אופן בדיוק לא נוכל לשלוח לשיטה מהצורה הזאת רשימה מסוג $List < String >$ למשל, כלומר הקוד הבא יגרום לשגיאה:

```
LinkedList < String > list = new LinkedList < String > ();
printList(list);
```

לעומת המקרה בו השיטה מוגדרת עם $List <?>$.

Wildcards extends:

ראינו מקודם שהקוד הבא גורר שגיאה:

$$LinkedList < Animal > myObjList = new\ LinkedList < Dog > (\quad);$$

לכן בעזרת $Wildcards\ extends$ נוכל לפתור את הבעיה הזו באופן הבא:

$$LinkedList <? extends\ Animal > myObjList = new\ LinkedList < Dog > (\quad);$$

(כלומר זה יעבוד לכל $LinkedList$ של אובייקטים שיורשים מ-$Animals$)

הערה חשובה:

נאתחל מצביע מהצורה: $List <? extends\ Animal > myList;$

(כלומר מצביע $myList$ שיחזיק בעתיד רשימה של אובייקטים שיורשים מ-$Animal$)

הפקודה $myList.add(new\ Dog());$ וגם הפקודה $myList.add(new\ Animal());$ יגררו שגיאת קומפילציה, כי עדיין לא הגדרנו לתוך $myList$, את סוג הרשימה הקונקרטי.

כפי שאמרנו כבר מקודם, הדבר היחיד שכן יתקמפל זה $myList.add(null);$ מכיוון שהדבר היחיד שיכול להכנס לכל סוג של רשימה הוא $null$.

הערה: בניגוד ל-$Generics$ שהם $Invariants$ (כפי שהסברנו מקודם), מערכים הם $Covariants$.

כלומר אין בעיה לעשות $Object[]\ objArray = new\ String[10];$ ולכן כדי למנוע בעיות, לא נוכל לייצר מערך של אובייקטים שמשתמשים ב-$generics$ (למשל לא ניתן לעשות $ArrayList[]$)

Erasure

לאחר קומפילציה, $Erasure$ "מוחק" את כל סוגי ה-$generics$ שהגדרנו,

והופך אותם לאובייקטים מאותו סוג. למשל האובייקטים $List < Animal >, List < Dog >$

שניהם יהפכו לאחר קומפילציה להיות $List$. הסיבה להתנהגות הזו היא ש-$generics$ זה קונספט יחסית חדש, שנכנס ב-$Java\ 5$ וכדי לתמוך בגרסאות קודמות של $Java$, קיים ה-$Erasure$.

$generics$ מוודא בזמן קומפילציה שאין שגיאות $Type - Error$, ולכן אם הקוד שלנו מתקמפל, $generics$ "עשה את שלו" ואין בעיה לבצע את התהליך של $Erasure$ כדי לתמוך בגרסאות קודמות.

השלכות של $Erasure$:

נסתכל על הקוד הבא:

$$ArrayList < String > I1 = new\ ArrayList < String > ();$$
$$ArrayList < Integer > I2 = new\ ArrayList < Integer > ();$$
$$System.out.println(I1.getClass() == I2.getClass()\ ;$$

המחלקה $getClass$ מחזירה ייצוג כלשהו של המחלקה של $I1$ או $I2$.

לכאורה, היינו מצפים שהתוצאה תהיה $false$ שכן $ArrayList < String > \neq ArrayList < Integer >$

אך בדיוק בגלל תהליך ה-$Erasure$, לאחר קימפול המחלקות הן $ArrayList = ArrayList$,

ולכן בפועל הפלט הוא $true$ (גם בזמן קומפילציה, הפלט יהיה $true$)

Regular Expressions

כלי שימושי מאוד לעבודה עם טקסט, בעזרתו נוכל למצוא / לקבל / לחפש / לטפל בטקסט שיש לו צורה מסוימת. למשל תאריך עם התבנית הבאה: $dd/MM/yyyy$ (01/05/2012)

בעזרת ביטוי רגולרי נוכל לוודא שהטקסט אכן מהצורה הנ"ל בצורה פשוטה.

<u>תווים רגולריים</u>:

Char	Usage	Example
a,b,c,...	Regular text	**abc** *matches* abc
.	Matches any single character	**.at** *matches* cat, bat, rat, 1at…
[…]	Matches any single character of the ones contained	**[cbr]at** *matches* cat, bat, rat.
[^…]	Matches any single character except for the ones contained	**[^bc]at** *matches* rat, sat…, but *does* ***not*** *match* bat, cat.
[a-z]	Matches any character in the range a-z Also works for A-Z, 0-9, and in the negative form (with ^)	- **[f-l]aaa** *matches* faaa, gaaa,…,laaa - **[^a-f]aaa** *matches* gaaa, 5aaa, &aaa, but *does* ***not*** *match* aaaa, caaa, ….
…	…	

נסביר את ה-2 הראשונים, והשאר יהיו ברורים יותר רק מהסתכלות בטבלה:

a, b, c – גם תווים בודדים הם ביטוי רגולרי, של עצמם. למשל $abc = abc$

. (נקודה) – מתאים לכל תו אפשרי. למשל ביטוי מהתבנית $.at$ הוא למשל $cat, bat, rat, 1at$.

<u>כמתים רגולריים</u>:

Char	Usage	Example
*	Matches zero or more occurrences of the single preceding character	- **.*at** *matches* everything that ends with *at:* at, hat, 123_$&treat… - **<[^>]*>** *matches* <…anything…>
+	Matches one or more occurrences of the single preceding character	**0+123** *matches* 0123, 00123, 000123…

$*$ (כוכבית) – 0 מופעים או יותר של ביטוי מסוים.

למשל הביטוי הרגולרי $.*at$. יתאים **כל** ביטוי שנגמר ב-$.at$.

למה? כי . (נקודה) אומר "כל תו", * (כוכבית) אומר 0 או יותר פעמים, ומכיוון שהיא באה אחרי נקודה, הכוונה היא "כל תו, 0 או יותר פעמים" כלומר כל ביטוי שהוא, וה-at מבטיח שהביטוי מסתיים ב-at.

למשל הביטוי הרגולרי $<*[^>]>$ יתאים כל ביטוי מהצורה $< \cdots anything \ldots >$

למה? הסוגריים המשולשים הקיצוניים $< >$ מציינים שאנחנו מחפשים ביטוי מהצורה $< \cdots >$,

פנים הביטוי הרגולרי $[^>]$ מבטיח שהסוגריים המשולשים לא כוללים את התו $>$

(כלומר הוא פוסל למשל את המקרה $< \cdots > \cdots >$ שיש 2 (או יותר) סוגריים משולשים סוגרים)

ולסיום ה-$*$ (כוכבית) מבטיחה שהביטוי מסתיים ב-$>$ ולכן נקבל ביטוי מהצורה $< \cdots anything \ldots >$.

+ (פלוס) - מופע אחד או יותר של ביטוי מסוים.

למשל הביטוי הרגולרי $0 + 123$ יחזיר ביטויים מהצורה 0123,00123,000123 אך לא את 123 .

דוגמאות:

1. איך נכליל ביטוי רגולרי לביטויים הבאים?

$rrrrrrabbit, rrabbit, rabbbit, rraabbiitt, rabbbit$ וכו'?

(כלומר המילה $rabbit$ באותיות קטנות, כך שמספר המופעים ברצף של כל אות הוא בלתי מוגבל)

כך: $r + a + bb + i + t +$ (נשים לב שרשמנו $bb +$ ולא $b + b +$ שניהם אותו דבר)

2. איך נכליל ביטוי רגולרי לביטויים הבאים?

$MyRabbit, HISRABBIt, RabBit, gOOdrABBIt$ וכו'?

(כלומר רצף של אותיות כלשהן, לאחר מכן המילה $Rabbit$ כך שכל האותיות במילה

למעט האות האחרונה t, יכולים להיות אותיות קטנות או גדולות)

כך: $[A-Za-z] * [Rr][Aa][Bb][Bb][Ii]t$

Recursive Structure: (ביטוי רגולרי המוגדר בצורה רקורסיבית)

'|' - התו האנכי, משמש כמו אופרטור "או", כפי שאנחנו רגילים. למשל $abc|xyz$ יי

תאים ביטויים מהצורה abc או ביטויים מהצורה xyz.

$(a|b)c$ – סוגריים עגולים, בדיוק כמו שנעשה תנאי $if((a \mid\mid b) \&\& c)$

כלומר a או b אך בכל מקרה c אחריהם, למשל $ac,\ bc$.

קיצורים:

על נקודה כבר הסברנו (כל תו אפשרי), ונסתכל עכשיו על קיצורים נוספים:

.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\v\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

את כל הקיצורים כמובן אפשר להשיג בדרכים נוספות (כפי שניתן לראות בטבלה),

אך מכיוון שמדובר בביטויים נפוצים, יש להם קיצורים שימושיים.

דוגמא נוספת:

3. איך נכליל ביטוי רגולרי לביטויים הבאים?

$1\ rabbit\ , one\ rabbit, X\ rabbits$ (כאשר X הוא מספר)?

כך: $(1|one)\ rabbit|\backslash d +\ rabbits$

סימוני גבולות:

^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary

למשל $^\backslash d$ יתאים לכל ביטוי שמתחיל בספרה.

למשל $^\backslash d\$$ יתאים אך ורק לספרות 0 עד 9 , שכן מדובר בביטוי שמתחיל בספרה ונגמר בה.

דוגמה:

נניח שאנחנו רוצים לעבור על מסמך מסוים ולחפש רק את המילה "$rabbit$", לא כולל $rabbits$ ועוד..

אז הביטוי הרגולרי המתאים הוא $\boldsymbol{\backslash b}rabbit\boldsymbol{\backslash b}$ כי השתמשנו ב-$\backslash b$ שמגדיר גבולות של מילה.

לכן עבור "$Some\ rabbits\ plays\ in\ the\ yard$", לא קיימות התאמות, שכן יש s אחרי $rabbit$.

(למה לא להשתמש פשוט ב-" $rabbit$ "? (עם רווחים בצדדים?) כי זה לא יתפוס מקרים כמו:

" $rabbit\ rabbit$ "," ... $rabbit$"," ... $rabbit$" לעומת $\backslash b$ שכן יתפוס)

כמתים:

•	X{n}	X occurs exactly n times	
•	X{n,}	X occurs n or more times	
•	X{n,m}	X occurs at least n but not more than m times	
•	X?	X is optional (it occurs once or not at all)	[⇔ X{0,1}]
•	X*	X occurs zero or more times	[⇔ X{0,}]
•	X+	X occurs one or more times	[⇔ X{1,}]

ראינו למעלה את השימושים של ה-2 האחרונים, אך מה אם אנחנו רוצים מספר מדויק של מופעים?

$X\{n\}$ יוודא ש-X מופיע בדיוק n פעמים

$X\{n,m\}$ יוודא ש-X מופיע לפחות n ולא יותר מ-m,

$X\{n,\}$ יוודא ש-X מופיע n פעמים או יותר

$X?$ אומר ש-X אופציונאלי, כלומר יופיע פעם אחת או בכלל לא.

דוגמא:

4. נמצא ביטוי רגולרי כדי לוודא שכל אות במילה $rabbit$ יוכפל לכל היותר פעם אחת, ובנוסף יכול להיות s בסוף המילה. (למשל $rrabbbit$ או $rabbbits$)

   כך: $r\{1,2\}a\{1,2\}b\{2,4\}i\{1,2\}t\{1,2\}s?$

<u>סוגי התנהגות של כמתים:</u>

נניח שאנחנו מעוניינים לחפש ביטוי מסוים בקובץ טקסט בעזרת הכמתים שראינו לעיל.
נוכל לשלוט על סוג החיפוש שיתבצע באופן הבא:

<u>$greedy$ (חיפוש חמדן)</u> – מתבצע בעזרת הכמתים הבאים (התנהגות ברירת המחדל של כמתים):

$$X? \quad , \quad X* \quad , \quad X+ \quad , \quad X\{n,\} \quad , \quad X\{n,m\}$$

יבדוק את כל המסמך מתחילתו עד סופו ואז יחזור אחורה אות-אות ויחפש את מה שביקשנו.
לדוגמה: עבור הקלט "$xgaxxxxga$", ועבור ה-$regex$ $ga*$. (כלומר כל ביטוי שמסתיים ב-ga)
החיפוש החמדן יחזיר שהוא מצא את הביטוי בתחום $[0,8]$
החיפוש התחיל בצד שמאל, התקדם עד סוף המילה, ואז חזר אחורה 2 צעדים ומצא ga כנדרש.

<u>$reluctant$ (חיפוש מינימליסטי)</u> – מתבצע בעזרת הכמתים הבאים (נוסיף סימן שאלה):

$$X?? \quad , \quad X*? \quad , \quad X+? \quad , \quad X\{n,\}? \quad , \quad X\{n,m\}?$$

יתחיל בלבדוק מחרוזת ריקה, לאחר מכן יתקדם אות אחת, לאחר מכן יתקדם לשנייה, וכן הלאה.
לדוגמה: עבור אותו קלט ועבור ה-$regex$ $ga*?$. (הפכנו אותו לחיפוש מינימליסטי)
נקבל בסוף החיפוש $[0,2]$ **וגם** $[3,8]$. החיפוש המינימליסטי בדק את המחרוזת הריקה ולא מצא ga,
אחר-כך התקדם אות אחת ולא מצא ga,לבסוף התקדם לאות השנייה ומצא ga כנדרש ב-$[0,2]$.
משם המשיך והתקדם אות אות עד שמצא ga בתו השמיני ולכן החזיר $[3,8]$

<u>$possesive$ (חיפוש רכושני)</u> – מתבצע בעזרת הכמתים הבאים (נוסיף פלוס):

$$X?+ \quad , \quad X*+ \quad , \quad X++ \quad , \quad X\{n,\}+ \quad , \quad X\{n,m\}+$$

דומה לחיפוש החמדן, אך לא חוזר אחורה. (ללא $backtracking$)
לדוגמה: עבור אותו קלט ועבור ה-$regex$ $ga+*$. (הפכנו אותו לחיפוש רכושני)
נקבל בסוף החיפוש שאין התאמה. למה? נתקדם עד לסוף המילה,
אין ga בסוף המילה, ולכן נחזיר שאין התאמה. (כי אנחנו לא חוזרים אחורה)

סיבות להעדיף את החמדן על הרכושני:

שימושי במקרים בהם יש חפיפה בין החלק הראשון לחלק השני.

למשל בביטוי הרגולרי $[a-z]+ing$ (מחפש מילים שמסתיימות ב-ing)

יש חפיפה כי ing הן אותיות שנכללות בתוך $[a-z]$.

סיבות להעדיף את הרכושני על החמדן:

במקרים בהם אין חפיפה, כלומר אין צורך לבצע $backtraking$ ולכן הרכושני יהיה יעיל יותר.

למשל: $[a-z]+[0-9]+$ (כל ביטוי שהוא רצף של אותיות ולאחר מכן רצף של ספרות)

אחרי שנגיע לסוף הביטוי, אם לא נתקלנו בספרות, בהכרח אין ספרות לפני כן כי מדובר באותיות.

סיבות להעדיף את המינימליסטי על החמדן:

שימושי במקרים בהם יש חפיפה בין החלק הראשון לחלק השני,

וגם אנחנו יודעים שהביטוי הראשון הוא קצר יחסית.

למשל: $[a-z]+xxxxxxxxxx$ (ביטוי מהצורה $axxxxxxxxxx$)

החיפוש המינימליסטי יעצור כבר באות הראשונה,

לעומת החיפוש החמדן שיתקדם עד הסוף ואז יחזור עד להתחלה.

סיבות להעדיף את החמדן על המינימליסטי:

במקרים בהם יותר מהיר ללכת עד הסוף ולחזור אחורה.

למשל: $[a-z]+ing$ (ביטוי מהצורה $xxxxxxxxxxxxxxxxxing$)

נתקדם עד הסוף ונחזור 3 פעמים אחורה.

"להתקדם" זה טוב יותר מ"לבדוק":

להתקדם עד סוף הביטוי ואז לחזור אחורה ולבדוק רק 3 צעדים אם נתקלנו ב-ing,

זה טוב יותר מלבדוק כל צעד עד הסוף אם נתקלנו ב-ing. כל בדיקה "עולה" זמן מסוים.

המטרה שלנו בהעדפת התנהגות מסוימת על אחרת היא מזעור כמות הבדיקות.

Capturing Groups:

נניח שאנחנו רוצים למצוא ביטוי שמכיל אות כפולה. למשל $letters$.

בעזרת הביטוי הבא: $([a-zA-z]*)([0-9]*)$ נוכל לבצע זאת.

כלומר, הוספנו סוגריים שתוחמות את הביטוי הרגולרי ובגדול "ממספרות" תת-ביטויים בו.

זאת אומרת שנוכל לגשת לערך שיוחזר מהביטוי $([a-zA-z]*)$ בעזרת $\backslash 1$, למשל:

$\backslash 1rabbit(\backslash d+)$ ימצא לנו ביטויים מהצורה $123rabbit123$ או $1rabbit1$

כי המספר שמצאנו על ידי $(\backslash d+)$ חויב להיות גם המספר בסוף המילה, על ידי $\backslash 1$

נקודות ליעילות $regex$:

- האופרטור "|" יכול להיות איטי לפעמים, לכן במקרים בהם נשתמש בו,
  נסדר את המחרוזות הפנימיות לפי סדר הופעתן: $(intro|oop|dast)$
  (בהנחה ש-$intro$ בעל הסבירות הגבוהה ביותר שיופיע, לאחר מכן oop ולאחר מכן $dast$)
- אם יש גורם משותף, נוציא אותו. למשל נחליף את $(abcd|abef)$ עם $ab(cd|ef)$
- $Backtracking$ זו נקודה כואבת של $regex$, נמנע מזה כל עוד אפשר,
  לכן נשתמש בחיפוש רכושני ולא בחמדן כל עוד ניתן.
- אם נרצה לתחום את הביטוי שלנו בסוגריים לטובת קריאות, אבל לא נרצה למספר אותם
  (כי זה "עולה" לנו למספר ביטויים), נוכל להשתמש ב-$(?:exp)$ שזה אומר שתחמנו את exp
  בסוגריים מבלי למספר אותם. למשל $(?:ab|cd)$ זה כמו לעשות $(ab|cd)$ רק בלי מספור.

מספור הסוגריים:

המספור הוא לפי הסוגריים הנפתחים, למשל בביטוי הבא: $(_1\ (_2\ A\ )\ (_3\ B\ (_4\ C\ )\ )\ )$

$$\backslash 1 = \big((A)(B(C))\big),\quad \backslash 2 = (A),\quad \backslash 3 = (B(C)),\quad \backslash 4 = (C)$$

רווחים משנים את הביטוי הרגולרי:

$[a-z]+[0-9] \neq [a-z]+[0-9]+$ כי $ab3 \neq ab\ \ 3$

regex ב-*java*:

נשתמש במחלקה $import\ java.util.regex.*;$

נגדיר $pattern$ חדש על ידי:

$$Pattern\ p = Pattern.compile("[a-z]+");$$

הערה: נשים לב שלא הגדרנו אובייקט חדש אלא השתמשנו במתודה סטטית של המחלקה.

במקרים בהם נרצה להשתמש באותו ביטוי $regex$ מספר פעמים,

נשתמש ב-$patt$ כמה פעמים ונמנע משימוש במתודה $compile$ כל עוד אפשר כי היא "יקרה".

לאחר מכן נגדיר $matcher$ חדש עבור קטע טקסט כלשהו:

$$Matcher\ m = parr.matcher("Now\ is\ the\ time");$$

למחלקה $Matcher$ יש 4 שיטות עיקריות:

$m.mathcer()$ – מחזיר $true$ אם"ם p מתאים **לכל** המחרוזת ($false$ אחרת)

$m.find()$ – מחזירה $true$ אם"ם p מתאים לחלק כלשהו במחרוזת.

(אם נקרא למתודה $find()$ שוב, היא תמשיך לחפש מאותו מקום שהפסיקה פעם קודמת.

אם היא תגיע לסוף הביטוי, היא תחזיר $false$ בכל קריאה)

$m.start()$ – יחזיר את האינדקס של התו הראשון בתת-המחרוזת/מחרוזת המתאימה.

$m.end()$ – יחזיר את האינדקס של התו האחרון בתת-המחרוזת/מחרוזת המתאימה.

למשל $srt.substring(m.start(), m.end())$ יחזיר לנו את תת-המחרוזת המתאימה.

אם ננסה לקרוא לשיטות $start, end$ לפני שהשתמשנו ב-$find$, תיזרק שגיאה.

- בתרגול 11 מוסבר גם על שיטות נוספות כגון $replaceFirst, replaceAll, reset$

שיטות נפוצות של המחלקה $String$ לשימוש עם $regex$:

Return value	Method name and description
boolean	*matches(String regex)* Tells whether or not this string matches the given regular expression
String []	*split(String regex)* splits this string around matches of the given regular expression
String	*replaceAll(String regex, String replacement)* Replaces each substring of this string that matches the given regular expression with the given replacement

$matches$ - מחזיר $true$ אם"ם ה-$regex$ הנשלח מתאים **לכל** המחרוזת ($false$ אחרת)

$split$ – מקבלת $regex$ ומחזירה מערך של מחרוזות לפי הביטוי הרצוי. למשל:

$$s = "hello \quad how \ are \quad you", \quad s.split("\ +");$$

כלומר פיצלנו את המחרוזת לפי רווח, והוספנו + כי לא ידוע כמה רווחים יש בין כל מילה.

נקבל בחזרה מערך מהצורה $["hello", "how", "are", "you"]$.

$replaceAll$ – מקבלת $regex$ ומחרוזת להחלפה, ומחליפה כל ביטוי שעומד בתנאי ה-$regex$ במחרוזת הנתונה. יכול להיות שימושי למשל לצנזורה.

השימוש בשיטות הקיימות של $String$ לעומת השימוש ב-$Pattern$ ו-$Matches$, הוא תלוי יעילות.

הקריאה ל-$Pattern.compile()$ היא פעולה יקרה, ומכיוון ש-$String.matches()$ קורא ל-$compile$ בכל קריאה מחדש, במקרים בהם נצטרך לקרוא ל-$String.matches()$ הרבה, נעדיף לייצר $Pattern$ אחד ולהשתמש בו מספר פעמים.

שאלנו בהתחלה איך אפשר לוודא שמחרוזת נתונה מייצגת תאריך מהצורה $dd/mm/yyyy$.

נוכל לעשות זאת כך:

```
boolean isDate(String s) {
        return s.matches("\\d{2}\/\d{2}\/\d{4}");
}
```

Functional Interface

בפיתון היינו יכולים לשלוח מצביעים לפונקציות בחתימות השיטה.

יכלנו לשלוח פונקציה $func1$ לפונקציה אחרת $func2$ ולהריץ את $func1$ מתוך $func2$.

ב-$Java$ קיימת פונקציונליות דומה אך היא טיפה יותר מסובכת למימוש.

<u>:$Runnable$</u>

אם נרצה להריץ פונקציית $void$ מסוימת,

נשתמש ב-$Interface$ המוכן $Runnable$ שמגדיר פונקציה אחת בלבד בשם $run()$,

וכל אובייקט שמממש את $Runnable$, אנחנו יודעים שיש לו שיטה run ונוכל להריץ אותה:

```
public void foo(Runnable r) {
        r.run();
}
public void doSomething1() {
        System.out.println("Doing it 1");
}
public void doSomething2() {
        System.out.println("Doing it 2");
}
```

```
public void bar() {
        foo(new Runnable() {
                public void run() {
                        doSomething1();
        }});
        foo(new Runnable() {
                public void run() {
                        doSomething2();
        }});
}
```

המתודה $bar()$ מפעילה פעמיים את המתודה foo, בכל פעם עם אובייקט $Runnable$ אחר.

הגדרנו את $Runnable$ בעזרת "מחלקה אנונימית" (דיברנו על זה בעמודים קודמים)

ובכל "מחלקה אנונימית" שהגדרנו, מימשנו את המתודה $run()$ שתריץ את $doSomething1/2$.

<u>:$Callable < T >$</u>

אם במקום להפעיל פונקציית $void$, נרצה לקבל ממנה ערך החזרה,

נשתמש ב-$Interface$ המוכן $Callable < T >$ שמגדיר פונקציה אחת בלבד בשם $call()$,

זו פונקציה גנרית ולכן היא יכולה להחזיר כל אובייקט שנרצה. נראה דוגמה בה נחזיר $Integer$:

```
public void foo(Runnable r) {
        try {
                int i = r.call();
        } catch(Exception e) {
                System.out.println("Exception.");
        }
}
public int doSomething1() {
        return 1;
}
public int doSomething2() {
        return 2;
}
```

```
public void bar() {
        foo(new Callable(Integer) {
                public Integer call() {
                        return doSomething1();
        }});
        foo(new Callable(Integer) {
                public Integer call() {
                        return doSomething2();
        }});
}
```

המתודה $bar()$ מפעילה פעמיים את המתודה foo, בכל פעם עם אובייקט $Callable$ אחר.

הגדרנו את $Callable$ בעזרת מחלקה אנונימית,

ובכל מחלקה כזו מימשנו את המתודה $call()$ שתחזיר את ערך ההחזרה של $doSomething1/2$.

Lambda

אם $Interface$ (לא מחלקה אבסטרקטית !) מגדיר רק מתודה לא דיפולטיבית אחת,
הוא יוגדר כ-$Functional\ Interface$. אינטרפייס פונקציונלי ניתן למימוש גם ע"י ביטויי $Lambda$.

<u>נראה דוגמות</u>:

איך זה נראה כביטוי $Lambda$:	איך זה נראה כפונקציה:
$() -> System.out.println("Doing\ it\ 1");$	$public\ void\ doSomething1() \{$ $System.out.println("Doing\ it\ 1");$ $\}$

- השיטה לא צריכה לקבל משתנים ולכן הסוגריים בתחילת ביטוי ה-$Lambda$ ריקים.

איך זה נראה כביטוי $Lambda$: $(x, y) -> x + y$ יכלנו לרשום גם כך, אך אין צורך: $(int\ x, int\ y) -> \{return\ x + y\}$	איך זה נראה כפונקציה:
	$public\ int\ doSomething2(int\ x,\ int\ y) \{$ $return\ x + y;$ $\}$

- אם ביטוי ה-$Lambda$ הוא שורה אחת, אין צורך לרשום באופן מפורש $return\ x + y$
אם מדובר בביטוי של יותר משורה אחרת, נהיה חייבים לציין במפורש.

איך זה נראה כביטוי $Lambda$:	איך זה נראה כפונקציה:
$foo(\ () -> doSomething()\ );$	$foo(new\ Runnable() \{$ $public\ void\ run() \{$ $doSomething();$ $\}\});$

- השיטה foo מצפה לאובייקט שמממש $Runnable$, לכן ביטוי ה-$Lambda$ "מתורגם"
למחלקה אנונימית, שמממשת את $Runnable$ עם השיטה **היחידה** $run()$,
שמוגדרת להפעיל את $doSomething()$.

איך זה נראה כביטוי $Lambda$:	איך זה נראה כפונקציה:
$public\ Comparator < Long > \ getComparator() \{$ $return\ (11, 12) -> (int)(2 * 11 - 12);$ $\}$	$public\ Comparator < Long > getComparator() \{$ $return\ new\ Comparator < Long > () \{$ $public\ int\ compare(Long\ 11, Long\ 12) \{$ $return\ (int)(2 * 11 - 12);$ $\}\};\}$

- כלומר במקום לממש מחלקה אנונימית בצורה מלאה, החזרנו ביטוי $Lambda$,
ו-$Java$ תדע "לתרגם" אותו למחלקה בדיוק כמו בשיטה השמאלית.

Serialization

תהליך תרגום/העתקה של אובייקטים, לפורמט שניתן לאחסן אותו (לדוגמה בקובץ) ולאחר מכן "להקים אותו לתחייה" באותה סביבת מחשוב או בסביבה שונה.
ב-$Java$ נבצע זאת ע"י "כתיבת" האובייקט ל-$Stream$. התהליך פועל באופן רקורסיבי, מכיוון שלאובייקט יכולים להיות משתנים שהם בעצמם אובייקטים, ולהם משתנים נוספים וכן הלאה.
התהליך ההפוך הוא $Deserialization$ שבו לוקחים אובייקט מ-$Stream$ ומשחזרים אותו להיות אובייקט באותו מצב שהיה לפני ה-$Serialization$.

איך $Serialization$ ממומש ב-$Java$?
ב-$java$, אובייקט צריך "להסכים" לזה שיוכלו לשמור אותו. (למשל מסיבות של מידע רגיש וכו') ברירת המחדל היא שאובייקט ב-$Java$ לא ניתן לסריאליזציה, וכדי לציין אחרת, עליו לממש את האינטרפייס $Serializable$. האינטרפייס הוא $Marker\ Interface$, כלומר אינטרפייס **ריק**, שכל מטרתו לסמן פונקציונליות / תכונה מסוימת. תהליך ההעתקה הוא רקורסיבי, כפי שהסברנו למעלה, לכן כל השדות (והשדות של השדות וכן הלאה) של האובייקט, צריכים גם הם לממש את $Serializable$. (שדות פרימיטיביים כגון $int, char, double$ וכו' הם ברי העתקה כברירת מחדל)

$Java\ Serialization\ Streams$:
נשתמש ב-$ObjectOutputStream$ כדי לבצע $Serialization$ לאובייקט,
וב-$ObjectInputStram$ כדי לבצע $deserialization$ לאובייקט.
2 המחלקות הן $decorating\ classes$ (מחלקות מקשטות) של מחלקות $stream$ אחרות.

דוגמה לביצוע סריאליזציה:

```
try(OutputStream out = new FileOutputStream("save.ser");
        ObjectOutputStream oos = new ObjectOutputStream(out); ) {
                oos.writeObject(new Date());
} catch(IOException e) {... }
```

- יצרנו $FileOutputStream$, ושלחנו אותו ל-$ObjectOutputStream$.
  (כאן בא לידי ביטוי העניין שמדובר במחלקה מקשטת)
- אחרי זה השתמשנו בשיטה $writeObject$ כדי לכתוב לקובץ.
- נזכור שצריך לתפוס את השגיאה $IOException$ כמו תמיד בשימוש בקבצים.

באופן דומה, דוגמה לביצוע דה-סריאליזציה:

```
try(InputStream in = new FileInputStream("save.ser");
        ObjectInputStream ois = new ObjectInputStream(in); ) {
                Date d = (Date) ois.readObject();
} catch(IOException e) {... }
```

- השתמשנו ב-$ObjectInputStram$, וההבדל היחיד הוא שאנחנו מקבלים מידע ולא כותבים מידע, ולכן הגדרנו משתנה $Date\ d$ שמקבל בחזרה את האובייקט (מסוג $Date$)
- בנוסף, ביצענו $Down - Casting$ לאובייקט המוחזר, כי $readObject$ מחזירה $Object$.

Object Graph in Object Streams:

נתחיל בדוגמה: נניח שיש לנו אובייקט שמייצג חשבון בנק משותף של בעל ואישה.
ל-2 האובייקטים שמייצגים את הבעל ואת האישה, יהיה שדה שמכיל מצביע לאותו חשבון בנק.
במקרה כזה, בזמן ביצוע סריאליזציה, ניתקל באובייקט שכבר ביצענו לו סריאליזציה,
(אם קודם ביצענו לגבר, אז אצל האישה ולהפך) ובמקרה כזה לא יתבצע התהליך שוב על האובייקט אלא נשמור מצביע לאובייקט הראשון שנשמר.
(כדי לחסוך במשאבים וזמן, וכדי למנוע לולאות אינסופיות בזמן התהליך)

נקודה חשובה: בגלל שנשמר רק מצביע ולא האובייקט כולו, כל שינוי שיתבצע מההעתקה הראשונה של אובייקט A (חשבון הבנק המשותף) לבין הניסיון השני להעתיק את A, לא יישמר.
למשל אם נעתיק את המחלקה של חשבון הבנק, לאחר מכן נשנה את הסכום בעובר ושב, ואז נעתיק שוב את המחלקה, בפועל בפעם השנייה אנחנו רק נשמור מצביע לאובייקט הראשון שהועתק ולכן השינוי לא בא לידי ביטוי בשום צורה. ניתן לעקוף את ה"מגבלה" הזאת ע"י קריאה לשיטה $reset()$ של המחלקה $FileOutputStream$ כדי שהיא "תשכח" איזה אובייקטים היא כבר העתיקה ואיזה לא, ובכך לכפות העתקה מחדש של האובייקט. (במחיר של בזבוז משאבים וזמן)

transient and static fields:

שדה שיסומן עם המילה השמורה $transient$, לא יישמר בתהליך הסריאליזציה,
ובתהליך ה-$deserialization$, השדות יקבלו את ערכם הדיפולטיבי.
ז"א שדות שמצביעים לאובייקטים וסומנו עם $transient$, יהיו $null$ לאחר דה-סריאליזציה.
$int, double$ יהיו 0 וכן הלאה. נרצה להשתמש ב-$transient$ על שדות שאין משמעות לשמור אותם, כמו למשל $streams$. באותו אופן, גם שדות סטטיים לא מועתקים בתהליך הסריאליזציה,
מכיוון שהם שייכים למחלקה כולה ולא לאובייקט ספציפי, ולכן אין משמעות להעתיק אותם.

נקודה חשובה: השיטה $writeObject$ מקבלת כקלט אובייקטים ולא פרימיטיביים (... $int, double$)
לכן כדי להעתיק פרימיטיביים נשתמש בשיטות ייעודיות כגון $writeInt, writeChar$ וכו'.

שינוי / עדכון מחלקות ששמרנו בזיכרון:

נניח ששמרנו מופע של אובייקט מסוים בזיכרון, ולאחר כחודש אנחנו מחליטים "להחזיר אותו לחיים".
בחודש שעבר ביצענו מספר שינויים במבנה המחלקה, הוספנו שדות, מחקנו שדות וכו'.
איך תהליך ה-$deserialization$ יתמודד עם השינויים? בעזרתנו.
לכל מחלקה שמממשת את $Serializable$, נגדיר את המשתנה הסטטי $SerialVersionUID$,
שיציין את הגרסה הנוכחית של המחלקה. זה המשתנה הסטטי **היחיד** שמועתק בתהליך
הסריאליזציה. נניח ששמרנו מופע של אובייקט מסוים בזמן שגרסת המחלקה הייתה 1,
ולאחר מכן במהלך כמה ימים ביצענו **שינויים מהותיים** במחלקה ושינינו את הגרסה ל-2.
בתהליך ה-$deserialization$, הגרסה הרשומה של האובייקט היא 1, הגרסה הנוכחית היא 2,
ולכן התהליך ייכשל. (וזה טוב, כי אנחנו רוצים למנוע בעיות)

שינויים מהותיים:

- שדות שונים (למשל שדה שהיה $string$ הוחלף להיות int)
- שינוי היררכיה של המחלקה

שינויים לא מהותיים במחלקה:

- מחיקת שדות (נוכל פשוט להתעלם מהשדה הזה בטעינה)

הערה חשובה: אם לא נגדיר את המשתנה באופן ידני, $Java$ תגדיר אותו בעצמה, אך כל שינוי (מהותי או לא מהותי) שנעשה במחלקה, יגרור עדכון של המשתנה, ובדיעבד יגרום לאי-תמיכה **בכל** הגרסאות הישנות של המחלקה, בכל שינוי קטן.

- מומלץ תמיד להגדיר את המשתנה ידנית בכתיבת מחלקה שמממשת את $Serializable$.

Cloning

שכפול אובייקטים. (שכפול פנימי, בלי להוציא אותו החוצה ל-$Stream$ כמו ב-$Serialization$)
באותו אופן כמו סריאליזציה, אובייקט צריך "להסכים" לכך שישכפלו אותו. כלומר אובייקט שניתן לשכפל אותו, צריך לממש את האינטרפייס $Cloneable$ (שגם הוא $Marker\ Interface$)
ולדרוס את השיטה $Clone()$ של המחלקה $Object$.

<u>קיימים 2 סוגי שכפול:</u>

$Shallow\ Copy$ – אם למחלקה יש שדות לא פרימיטיביים (אובייקטים),
רק המצביעים שלהם (ולא האובייקטים עצמם) יועתקו.
לכן השדה של אובייקט A והשדה של אובייקט A' (המשוכפל) הם אותו אובייקט בזיכרון !
כל שינוי באחד מהם יגרור שינוי גם בשני (כי בעצם מדובר באותו אחד)
$Deep\ Copy$ – שדות לא פרימיטיביים (אובייקטים) מועתקים באופן רקורסיבי,
כלומר נוצרים אובייקטים חדשים בזיכרון שמכילים את אותם ערכים,
ולכן שינוי שיתבצע בשדה אחד לא יגרור שינוי גם בשני.

<u>הערה חשובה</u>: המימוש הדיפולטיבי של השיטה $Clone()$ הוא לייצר $Shllow\ Copy$,
לכן במקרה בו נרצה $Deep\ Copy$, נדרוס את השיטה ונממש אותה בעצמנו.

<u>המתודה $Object.clone()$:</u>

- מוודא שהאובייקט הנתון מממש את $Clonable$, אם לא, תיזרק שגיאת $CloneNotSupportedException$.
- מבצעת $Shallow\ Copy$ באופן דיפולטיבי.

<u>הערה</u>: מערכים מממשים את $Clonable$, אך השכפול הוא שכפול שטוח ($Shallow\ Copy$).

<u>נראה דוגמה למימוש של $Clonable$:</u>

```
class Pet implements Cloneable {
        private Date birthDate;
        public Object clone() throws CloneNotSupportedException {
                // First – creating a shallow copy
                Pet pet = (Pet) super.clone();
                // Cloning date for deep copy
                pet.birthDate = (Date) birthDate.clone();
                return pet ;
        }
…..
}
```

- ייצרנו מחלקה pet שמממשת את $Clonable$ ודרסנו את $clone$ כדי לבצע $Deep\ Copy$.
- נשים לב שהשיטה $clone$ מחזירה $Object$ ולא Pet, לכן צריך לבצע $Down-Casting$ בכל פעם שנרצה לבצע שכפול לאובייקט מסוים. (כפי שעשינו ל-Pet ול-$Date$)
- בשורה הראשונה של השיטה ייצרנו עותק שטוח של האובייקט,
ולאחר מכן עדכנו את השדה $birthDate$ להיות אובייקט חדש ע"י העתקה עמוקה.
(יצאנו מנקודת הנחה שהמחלקה $Date$ מממשת $clone$ עם העתקה עמוקה)

<u>נראה דוגמה לשכפול אובייקטים:</u>

```
try{
        Pet myPet = new Pet();
        myPet.setType("Dog");
        Pet myPet1 = (Pet) myPet.clone();
        Pet myPet2 = (Pet) myPet.clone();
        myPet1.setName("Woofi");
        myPet2.setName("Goofi");
……
} catch(CloneNotSupportedException e) {
        e.printStackTrace(); // Checked Exception
}
```

- נזכור שאם אחד האובייקטים לא מימש את $Clonable$, תיזרק שגיאה ולכן נתפוס אותה.
- יצרנו Pet חדש, שינינו את הסוג שלו להיות "Dog", ולאחר מכן שכפלנו אותו פעמיים.
אחרי זה הגדרנו שמות ל-2 האובייקטים.

Copy Constructor

שימוש באינטרפייס $Clonable$ זו לא השיטה המומלצת לשכפול אובייקטים (כן, עוד אחת מהחפירות שהוא מביא רק כדי שנכיר את הדרכים הפחות טובות). פתרון טוב יותר יהיה לממש בנאי העתקה.
הפתרון הזה פשוט יותר, ומאפשר בנוסף שכפול של אובייקט מסוג מסוים לסוג אובייקט אחר.
למשל לשכפל $ArrayList$ לתוך $LinkedList$.

<u>נראה דוגמה למימוש בנאי העתקה:</u>

```
class Pet implements Cloneable {
        private Date birthDate;
        public Pet(Pet other) {
                this(); // First – calling default ctor.
                // Date class doesn't have a copy constructor
                // Use cloning instead
                this.birthDate = other.birthDate.clone();
        }
….
}
```

- הגדרנו בנאי שמקבל אובייקט כלשהו (במקרה הזה אובייקט Pet אחר), ואנחנו מעתיקים את המידע מהאובייקט הזה.
- $this()$ מקביל לקריאה $super.clone()$ שעשינו בדוגמה הקודמת.
- העתקנו את השדה $birthDate$ בעזרת $clone()$, שוב בהנחה שהיא מבצעת $Deep\ Copy$.

<u>נראה דוגמה לשימוש בבנאי העתקה:</u>

```
Pet myPet = new Pet();
myPet.setType("Dog");
Pet myPet1 = new Pet(myPet);
Pet myPet2 = new Pet(myPet);
myPet1.setName("Woofi");
myPet2.setName("Goofi");
```

- ייצרנו אובייקטים חדשים בעזרת בנאי ההעתקה.
- חסך לנו את הבדיקה של $try - catch$.
- יכלנו באותה מידה להגדיר בנאי העתקה לאובייקטים אחרים, לאו דווקא Pet, כמו למשל לייצר $ArrayList$ חדש שמעתיק ערכים מ-$LinkedList$.

Reflections

השם $Reflection$ מתאר קוד שיכול לבדוק קוד אחר באותה מערכת.

מדובר בכלי עוצמתי מאוד (ומסוכן!) שמאפשר למשל לגשת לכל המשתנים של מחלקה מסוימת לכל השיטות, לכל הבנאים וכו'. (אפילו אם הם מוגדרים כ-$private$)

נזכור שכל אובייקט ב-$Java$ הוא פרימיטיבי או רפרנס:

$Reference\ types$: יורשים מ-$Object$, למשל מחלקות, מערכים, אינטרפייסים ..

$Primitive\ types$: למשל $boolean, byte, char, double$ וכו'.

<u>המחלקה $Class$:</u>

לכל אובייקט מסוג רפרנס, מאותחלת בנוסף מחלקה מסוג $Class$.

נוכל לקבל את המחלקה הזו על ידי שימוש במתודה $forName$.

נניח שאנחנו רוצים לקבל את המחלקה $Class$ שנוצרה עבור מחלקה $MyClass$ כלשהי,

אז נשתמש ב: $Class\ cls = Class.forName("MyClass");$

כלומר מספיקה מחרוזת שמכילה שם של מחלקה, כדי לקבל את המחלקה $Class$ הרצויה.

(תיזרק שגיאת $ClassNotFoundException$ אם לא קיימת מחלקה כזו)

אם יש לנו מצביע לאובייקט עצמו, נוכל לעשות: $Class\ cls = myObj.getClass();$

נראה עכשיו מספר דוגמאות לשימוש של $Reflections$.

<u>$getDeclaredConstructors()$:</u>

המתודה $getDeclaredConstructors$ מאפשרת לנו לקבל את כל הבנאים של מחלקה מסוימת:

$$Constructor[]\ ctorlist = cls.getDeclaredConstructors();$$

יצרנו מערך של בנאים, שמכיל את כל הבנאים של המחלקה.

<u>$newInstance()$:</u>

המתודה $newInstance$ מחזירה אינסטנס חדש של המחלקה בעזרת הבנאים שקיבלנו:

$$Object\ retobj = ctorlist[i].newInstance(arglist);$$

כאשר $arglist$ הכוונה לארגומנטים שאמורים להישלח לבנאי.

אפשר לקבל אותם בעזרת השיטה $getParameterTypes()$ של הבנאי.

<u>$getDeclaredMethods()$:</u>

המתודה $getDeclaredMethods$ מחזירה מערך של מתודות של המחלקה. (כולל מתודות פרטיות!)

$$Method[]\ methlist = cls.getDeclaredMethods();$$

אפשר להשתמש במתודות של המחלקה $Method$ למשל:

$getDeclaringClass()$ – איזה מחלקה הגדירה את המתודה.

$getParameterTypes()$ – רשימה של סוג הפרמטרים שהם מקבלים.

ושיטות נוספות..

invoke():

המתודה *invoke* מאפשרת להריץ מתודות של המחלקה.

```
methlist.[j].invoke(obj, arglist);
```

כאשר *obj* זה האינסטנס של המחלקה שייצרנו מקודם ו-*arglist* רשימה של ארגומנטים לשיטה.

ערך ההחזרה של השיטה יהיה *Obj* , או *null* אם מדובר בשיטת *void*.

getDeclaredFields():

המתודה *getDeclaredFields* מחזירה את השדות של המחלקה.

```
Field[] fieldlist = cls.getDeclaredFields();
```

אפשר להשתמש במתודות *set\get* עם השדות האלה ולשנות אותם:

```
Field[i].set(Object obj, Object data)   \\   Field[i].get(Object obj)
```

ברירת המחדל היא שלא ניתן לגשת לשדות *private*,

וניסיון לגשת אליהן יגרור שגיאת *IllegalAccessException*.

כדי בכל זאת לגשת לשדות *private*, נוכל להשתמש ב-*Field[i].setAccessible(true);*

כדי לאשר גישה, ואז לבצע את ה-*set\get* הרצוי.

נסתכל על דוגמה לקוד שמקבל מחרוזת עם שם של מחלקה,

ומדפיס את כל השמות של השדות והערכים שלה:

```
import java.lang.reflect.*;
public class DumpMembers {
        public static void main(String args[]) throws ClassNotFoundException,
                                InstantiationException, IllegalArgumentException,
                                IllegalAccessException, InvocationTargetException {
        Class cls = Class.forName(args[0]);
        Field[] fields = cls.getDeclaredFields();
        Constructor[] ctors = cls.getDeclaredConstructors();
        Object obj = ctors[0].newInstance();
        for(Field field: fields)
                if(Modifier.isPublic(field.getModifiers()))
                        System.out.println(field.getName() + ": " + field.get(obj));
        }
}
```

למה $Reflections$?

- גמישות –למשל במחלקות $Factory$ שעשינו ב-$ex5$, היינו צריכים לציין ידנית את סוגי הפילטרים בעזרת תנאים או $cases$. בעזרת $Reflections$ יכלנו לממש קוד כללי בלי כל הפירוט, שבנוסף גם לא ידרוש עדכון לאחר הוספת פילטרים חדשים.
- דיבוג – למשל $JUnit$ מבצע שימוש ב-$Reflections$.
- $Serialization$ – איך המחלקה $ObjectOutputStream$ מעתיקה את כל המשתנים, המתודות והבנאים? (אפילו הפרטיים?) בעזרת $Reflections$.

חסרונות של $Reflections$:

- נוגד את העקרונות שלמדנו במהלך כל הקורס ($encpsulation$ וכו')
- יכול לגרום לבעיות בקוד או לפגוע ביכולת להעביר את הקוד מפלטפורמה לפטלפורמה.
- תוכניות שמשתמשות ב-$Reflections$ הן איטיות יותר.

לסיכום: $Private\ is\ not\ Secret$

כלומר אם יש מידע רגיש כמו סיסמאות, כרטיסי אשראי וכו', אחסון שלהם תחת משתנה $private$ זה לא הפתרון בדיוק מהסיבה ש-$Reflections$ קיים. כדי לאחסן מידע רגיש, נצפין אותו.

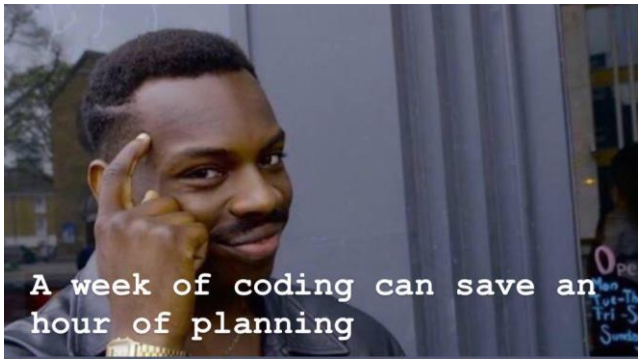
הסיבה שאנחנו בכל זאת משתמשים ב-$private$ היא בשביל $design$ טוב יותר.

אנחנו מתחייבים ללקוח ל-API מסוים, ומבטיחים לו שאם הוא יסתמך עליו הקוד שלו יעבוד תמיד.

אם מישהו ישתמש ב-$Reflections$ כדי לגשת לשיטות $private$ שלנו, ובעתיד אנחנו נחליט למחוק אותן, זו בעיה שלו ולנו אין כל התחייבות אליו במקרה הזה.


קוד של השותף


BIRTH CONTROL EFFECTIVENESS
CONDOMS
99%
BIRTH CONTROL PILLS
99%
מדמח שנה א'
100%


A week of coding can save an hour of planning


99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.
127 little bugs in the code...


LEFT
EXIT 12
להשתמש בדיבאגר
לעשות 10000 הדפסות
אני


בפייתון זה לא היה קו..
שתוק!
קראוזה