**תרגול OOP – שבוע 7**

**מחלקה מקוננת:**

* מחלקה שמצהירים עליה בתוך מחלקה אחרת
* יש שני סוגים:
  + Nested static class – נצהיר עליה כמחלקה סטטית static nestedClass
  + Inner class – לא מצהירים עליהם בתור סטטיות
* מחלקה מקוננת היא member של המחלקה
* אם היא לא סטטית יש לה גישה לכל מה שיש במחלקה החיצונית (גם פרטיים), לעומת סטטיות שאין להם גישה לשום דבר כי הן מקושרות רק למחלקה העוטפת בכללותה.
* אפשר להצהיר עליהן עם כל אחד מהמודיפיירים – פומבי, פרטי, מוגן
* למה להשתמש במחלקות מקוננות?
  + דרך לרכז מחלקות שמשתמשים בהם רק פעם אחת – למשל מחלקות עזר.

עוזר ליעל ולארגן, אם הן משומשות למשהו ספציפי

* + עוזר לאנקפסולציה – אם נרצה שמחלקה אחרת תשתמש בה אבל לא שכולם יראו את הנתונים שלה
  + מאפשר לרכז את הקוד שמקום שהוא קורה במקום לחלק את הקוד בין קבצים
* Inner classes – מחלקות מקוננות לא סטטיות
  + מקושרות למופע מחלקה ספציפי של המחלקה ויש לה גישה לכל השדות והמתודות, גם הפרטיים, של המופע.
  + לא אמורים לרשת מחלקה פנימית אם לא ירשנו מהחיצונית, בדומה אי אפשר ליצור מופע ספציפי שלה כי היא מקושרת למופע של המחלקה החיצונית.
* Nested static classes:
  + מקושרות למחלקה החיצונית ולא מקושרות למופע ספציפי
  + אין לה גישה לשדות ומתודות של מופעי מחלקה חיצונית
  + אפשר לחשוב עליה כמו כל מחלקה אחרת רק שבחרנו להכניס אותה לתוך מחלקה אחת – אין לה יתרונות של גישה, זה רק עבור סדר וריכוז
  + אפשר ליצור אובייקט של המחלקה עצמה כי היא לא מקושרת למופע מסוים

**ממשקים:**

* ניתן להכניס מתודות פרטיות
* אם נרצה להוסיף לממשק עוד מתודות –
  + אפשר לרשת ממשק זה ולהוסיף לו עוד מתודות
  + להוסיף מתודות דיפולטיות (מחייב להוסיף מימוש שלהן)
  + להוסיף מתודות סטטיות (מחייב מימוש שלהן)
* לשתי האפשרויות האחרונות לא נצטרך לקמפל מחדש את המחלקות שממשות את הממשק – כלפי חוץ נישאר אותו דבר

**מתודות דיפולטיות:**

* נגיד שיש לנו ממשק שכולל מתודה דיפולטית ואנו יורשים ממנו –
  + אם לא נגדיר את המתודה הדיפולטית יהיה לנו את המימוש הדיפולטי שלה
  + להצהיר על המתודה הדיפולטית מחדש בלי המילה דיפולט ואז היא תהפוך להיות אבסטרקטית – מי שיממש את הממשק יצטרך לממש אותה (מסיר את הדיפולטיות שלה)
  + לדרוס אותה כמתודה דיפולטית ונצטרך לממש אותה
* אם יש לנו מתודה דיפולטית בממשק אבל גם מתודה עם אותו שם במחלקה שמממשת אותו, לאיזו מתודה נפנה?

3 חוקים שעוזרים לקביעה לאיזה מתודה נלך:

* + אם יש הצהרה על מתודה בתוך המחלקה המממשת אז מה שיש בתוך הממשקים לא מעניין אותנו
  + אם יש לנו ירושת ממשקים אז המימוש שנלך אליו הוא המימוש הכי תחתון בעץ הירושה
  + במקרה שיש משהו שנראה כמו ירושת יהלום, כלומר אם ירשנו (מימשנו) שני ממשקים עם אותה מתודה נצטרך להגיד באיזה מהם נרצה להשתמש – אם ירשנו מA ומB כדי לגשת למתודה נכתוב A.super.name()

**מתודות סטטיות:**

* ניתן להגדיר מתודות סטטיות בממשקים
* נצטרך להצהיר עליהן כסטטיות ונצטרך לממש אותן.
* בעיקר בשביל פונקציות עזר

**מתודות פרטיות:**

* ניתן להוסיף מתודות פרטיות לממשקים (לא תהיה אליהן גישה מבחוץ)
* מאפשרות למנוע כפל קוד בין מתודות דיפולטיות או סטטיות בממשקים

**מחלקות אבסטרקטיות וממשקים:**

* מחלקות אבסטרקטיות דומות לממשקים – אי אפשר ליצור מהן מופע
* מה יש למחלקות אבסטרקטיות ולא לממשקים:
  + ניתן להגדיר בהן שדות שהן לא static / final
  + ניתן להגדיר מתודות פומביות, פרטיות או מוגנות
* בממשקים כל השדות הם אוטומטית פומביים, סטטיים וסופיים, וכל המתודות אוטומטית פומביות
* מתי נעדיף להשתמש במחלקה אבסטרקטית?
  + לשתף קוד בין הרבה מחלקות יחסית קרובות
  + מצפים לרשת מהמחלקה האבסטרקטית שירחיבו אותה אבל יש להם מתודות ושדות משותפים
  + להגדיר שדות לא סטטיים או סופיים
* מתי נעדיף ממשקים?
  + כשנרצה להגדיר כלל או התנהגות ברמה הכי כללית, בלי לתת מידע לדוגמא ספציפית
  + לאפשר ירושה מכמה מקומות שונים (ניתן לממש כמה ממשקים במקביל)
  + אם מחלקות שאין בהן בהכרח קשר ירצו לממש את הממשק – המחלקות לא קשורות אבל יש ביניהן קשר מסוים
  + אם נרצה להגיד משהו על התנהגות הטיפוס אבל לא אכפת לנו מי ממש אותה ואיך.
* ממשקים עוסקים יותר בהצהרה של תכונות כלליות שצריכות להיות לטיפוס שלנו בעוד שמחלקות אבסטרקטיות יותר מדברות על דוגמאות לטיפוס.

**תבנית עיצוב – איטרטור:**

* נניח שנבנה טיפוס מסוג אגריגט (מכיל רפרנסים לסט של אובייקטים), נרצה אפשרות לעבור על האובייקטים שמחזיק.
* שתי דרכים:
  + להגדיר לאגריגט עצמו מתודות של next ו-previous
  + לספק איטרטור – אובייקט שמאפשר לנו לעבור על האיברים באגריגט בצורה שנרצה
* האיטרטור והאגריגט שמקושר אליו הם מאוד קשורים – איטרטור ספציפי לאגריגט, המימושים שלהם מושפעים.
* נצטרך לכתוב את האיטרטור בתוך האגריגט (סוג של מחלקה מקוננת) – האיטרטור צריך להכיר את המימוש הפנימי של האגריגט וצריך גישה לשדות שלאו דווקא האגריגט נותן גישה מבחוץ.
* איך נראית תבנית העיצוב – מורכבת מ4 חברים:
  + איטרטור – ממשק עבור מעבר על איברים
  + מחלקה של איטרטור קונקרטי - מממשת את הממשק
    - שומרת מידע על המיקום במעבר, כלומר מה האיבר הבא שנעבור עליו
  + ממשק של אגריגט שמאפשר יצירת איטרטור (כלומר למעשה יוצר ממשק איטרטור
  + אגריגט קונקרטי ממש את הממשק ויוצר גם את האיטרטור הקונקרטי
* הדרך הטובה ביותר ליצור איטרטור היא בעזרת מחלקה פנימית – Inner class –
  + מאפשר גישה לכל המידע שנצטרך על האגריגט
* מעבר על רשימה מקושרת שונה ממעבר על רשימה רגילה
* בג'אווה הוא לא משהו שקל לרשת ממנו בפני עצמו (כלומר אפשר לרשת מאיטרטור מבלי לרשת מהמחלקה עצמה אבל זה מסורבל)

אבל תבנית העיצוב יחסית קלה למימוש בהנחה שאנו מכירים את

* בספרייה הסטנדרטית של ג'אווה יש את Iterable ואת iterator
  + Iterable ממשק למחלקה ש/ מחזיקה איטרטור
  + אפשר להשתמש בממשקים המוגדרים האלה במקום ליצור ממשק חדש לאיטרטור ואגריגט
* דוגמא:
  + ממשק איטרטור – מתודה שמחזירה את האובייקט הבא ומתודה ש
  + ממשק אגריגט – מחזיקה מתודה ליצירת איטרטור ומתודה שמחזירה אמת אם הצלחנו להוסיף איבר ושקר אחרת
  + האגריגט – מעטפת למערך שמממשת את שתי המתודות
  + מחלקה פנימית איטרטור – מחזיקה סמן שמאותחל ל-0, בודקת אם יש עוד איבר בעזרת השדה גודל שהאגריגט, ומחזיקה מתודה next שמחזירה את האיבר הבא ומקדמת את הסמן.