**פרק 6 – OOP**

**6.1** **מונחה עצמים – שדות סטטיים:**

* **מה זה שדה סטטי?**
  + קבוע – ההגדרה מתחילה ב-3 מגדירים:
    - Public
    - Static
    - Final
  + Static – שדה שלא מייצג תכונה של המופע
    - יש רק עותק אחד שלו עבור כל מופעי המחלקה
  + משתנים סטטיים (אלא אם מדובר בקבועים) הם לא מומלצים,
  + תכונה שקשורה לקטגוריה, כלומר למחלקה, היא תכונה שמיוצגת כשדה סטטי
  + לשדה סטטי קוראים גם משתנה מחלקה, class variable
    - לעומת שדה רגיל שקוראים לו גם משתנה מופע, instance variable
  + לשדות סטטיים ניגשים כמו קבועים – עם שם המחלקה נקודה שם השדה הסטטי
* **שדות סופיים – final:**
  + Final אומר שהמשתנה מקבל ערך רק פעם אחת בריצת התכנית.
  + כלומר אי אפשר לעדכן את ערך המשתנה אם הוא סופי – שגיאת קומפילציה
  + נקודות חשובות:
    - שדה סופי חייב לקבל ערך או בבנאי או בשורת האתחול שלו (לא בשניהם!)
    - שדות סופיים פרטיים – מגנה מפני שינוי של השדה שנעשה בתוך העצם
    - רפרנסים סופיים – אם שדה מצביע הוא סופי – העצם אליו מצביע יכול להשתנות אבל העצם אליו אנו מצביעים לא משתנה, כלומר הרפרנס לא יכול להשתנות.
  + Mutability –
    - Mutable – ניתן לשינוי
    - Immutable – בלתי ניתן לשינוי
  + קבוע הוא שדה סטטי וסופי –
    - בגלל שהוא סופי הוא יכול לקבל ערך או בבנאי או בשורת ההצהרה
    - בגלל שהוא סטטי הוא לא קשור לאף מופע ולכן יוכל לקבל ערך רק בשורת ההצהרה.
  + מצביע לעצם מיוטבילי לא נחשב קבוע – הרפרנס הוא הקבוע אבל את העצם עצמו ניתן לשנות
  + אם השדה הסופי הוא מצביע השדה יחשב לא מיוטבילי כי את המצביע עצמו לא נוכל לשנות
* **העשרה:**
  + לא ניתן ליצור מופעים של Math.
  + הדבר נעשה על ידי הגדרת בנאי פרטי - מכיוון שמוגדר בנאי (הגם שהוא פרטי), לא נוצר במחלקה בנאי ברירת מחדל, ולכן אין ל-Math בנאי פומבי.
  + האם לא ניתן לרשת מ-Math ואז להגדיר למחלקת הבת בנאי פומבי, וכך ליצור בעקיפין מופעים של Math?
    - לא. המחלקה Math משתמשת במשמעות אחרת של המילה השמורה final:

public final class Math {…}

* + - בהקשר כזה, משמעות המילה final היא שלא ניתן לרשת מהמחלקה.
    - באופן דומה, שיטה שמוגדרת כ-final לא ניתנת לדריסה על ידי מחלקת הבת.
  + כלומר אם נגדיר מחלקה כ-final לא יהיה ניתן לרשת ממנה.

אם נגדיר שיטה כ-final, מחלקה שיורשת לא תוכל לדרוס אותה.

* **בעיות במיוטביליות משותפת – Shared mutability:**
  + מיוטביליות של משתנים קובעת את מספר החלקים "הנעים" בתוכנה.
  + מיוטביליות פחות מפריעה ככל שהיא משותפת לפחות חלקים בקוד – נרצה להימנע ממיוטביליות משותפת מאוד.
  + עבור משתנים מיוטביליים מקומיים יחסית פשוט לעקוב אחרי השינויים שלהם בתוך השיטה בה הם נמצאים.
  + כאשר נדבר על שדות אנחנו מדברים על כל השיטות שעקרונית יכולים לבדוק ולשנות את ערך השדה
    - ושדה מיוטבילH PUNCH JAU; DO KAHBUHHO NCJU./
  + Mutable field – שדות מיוטביליים, גם אם פרטיים, הם מקרו למיוטביליות משותפת.
    - שדות פרטיים מיוטביליים הם מקור למיוטבילית משותפת בתוך העצם.
  + כדי להקטין היקף של מיוטביליות משותפת בתוך העצם נרצה להפוך שדות מיוטביליים שדות אימיוטביליים – אם הם פרימיטיביים זה קל, נוסיף להם final.
  + שדות סטטיים:
    - שדה סטטי מיוטבילי נגיש משיטות כל מופעי העצם ולכן קשה לעקוב אחרי השינויים שקורים בו כי זה יכול לקרות מכל מופע מחלקה.
      * כלומר במקרה כזה יכול להיות שעצמים משפיעים אחד על השני בצורה שיהיה לנו קשה לדעת כלקוחות.
    - שדות סטטיים מיוטבילים הם מכשולים של בדיקות אוטומטיות – אין לנו שליטה מלאה על מצב העצם ומה שמשפיע עליו אם יש לנו שדה סטטי מיוטבילי ואם אין לי שליטה כזו אז הטסטביליות של המחלקה נפגעת.
  + הבעיה הכי גדולה עם משתנים סטטיים היא שאנו מגבילים את עצמנו – הוא מייצג את כל המופעים ולכן זה מאוד מחייב, כלומר תמיד ישרת את כל מופעי המחלקה גם אם נרצה לחלק אותם.
    - איזה תחליף אפשר לתת?

שדה סטטי שמשויך לכל המופעים ניתן להחליף במופע של מחלקה אחרת שמשויך לקבוצה של מופעים

* + - * כלומר כמה מופעי מחלקה יצביעו לאותו עצם שמשותף להם במקום השדה הסטטי
      * אפשרי לשימוש גם אם המופע יהיה בסוף משותף לכל המופעים
  + יותר קל לכתוב תוכנה שבה המיוטביליות פחות משותפת.
* **העשרה:**
  + בהינתן כל מה שדיברנו עליו, אולי ישמע מפתיע ששדה מיוטבילי סטטי פומבי יהיה דפוס תכנותי נפוץ, אבל "תבנית העיצוב" Singleton מתייחסת בדיוק לזה.
  + **סינגלטון** הוא תבנית שבה מגדירים עצם מיוטבילי שיהיה זמין בקלות מכל מקום בתכנה. הנה למשל הדרך הפשוטה ביותר להגדיר סינגלטון:

class MutableClass {...

public static final MutableClass SINGLETON = new MutableClass();

…}

* + כעת, מכל מקום בקוד, ניתן לכתוב MutableClass.SINGLETON ולגשת לשיטות של העצם.
  + השם Singleton מגיע מכך שזהו גם המופע היחיד של המחלקה: בדרך כלל המתכנתים יחסמו את האפשרות להגדיר מופעים אחרים של המחלקה.
  + הסינגלטון זכה לפרסום כתבנית-עיצוב למקרים בהם נראה שיש צורך רק במופע אחד של המחלקה, ורוצים להקל על הגישה אליו.
  + כיום סינגלטון נחשב אנטי-תבנית-עיצוב (קרי: דפוס תכנותי נפוץ, אבל לא לפתרון בעיות כי אם ליצירה שלהן).

**6.2 שיטות סטטיות:**

* **מהן שיטות סטטיות?**
  + שיטות רגילות הן למעשה פעולה של העצם.
  + אם יש לנו שיטה שאנו רוצים שתבדוק משהו עבור על המופעים – היא לא קשורה למופע ספציפי של עצם.
  + שיטות סטטיות לא משויכות לעצם, אלא משויכות למחלקה לפי עקרון ה-dicoverability
  + נוסיף את המילה השמורה static לאחר מגדיר הנראות.
  + קריאה לשיטה סטטית מתוך המחלקה היא כמו קריאה לכל שיטה, כלומר שם השיטה.
  + קריאה לשיטה סטטית פומבית מחוץ למחלקה היא עם שם המחלקה נקודה שם השיטה.
  + לשיטה סטטית אין גישה ל-this ולאיברי מופע רגילים (שדות ושיטות לא סטיים)
  + שיטה סטטית יכולה לגשת לשדה סטטי.
* **דוגמאות לשיטות סטטיות:**
  + שיטה סטטית נשים תחת המחלקה שהכי קשורה אליה, אם מחלקה זו היא ממשק זה גם אפשרי (לא מאוד נפוץ).
    - שיטה סטטית בממשק היא כמו כל שיטה סטטית אחרת עם מימוש והכל ואין לה ממש קשר לתפקיד העקרוני של ממשק בפולימורפיזם.
    - ממשק מייצג איזשהו חוזה שבא לידי ביטוי בשיטות האבסטרקטיות שלו, השיטה הסטטית כנראה קשורה למופעים שלו אבל היא לא חלק מחוזה הממשק
  + שימושים בשיטה סטטית פומבית:
    - פונקציות פומביות שלא שייכות לאף עצם –
      * בפרט, שיטה שמייצרת מופע של המחלקה בה היא נמצאת – **שיטת מפעל** (כמו String.format/join, בדר"כ במקום בנאי בשביל API נוח וברור יותר)
    - שיטות עזר פרטיות שלא תלויות בשדות (לא נודעות למצב העצם)
      * למה שנרצה להשתמש בשיטה סטטית? שדות מהווים מיוטביליות משותפת בתוך המחלקה לכל השיטות הרגילות. אם ניתן להגדיר שיטה שלא נוגעת לשדות נוכל להקטין כך את מספר החלקים הנעים – שיטה סטטית מבהירה שהשיטה לא יכולה לגעת בשדות ספציפיים של אף מופע מסוים.
    - שיטות עזר לשיטות סטטיות פומביות
      * בפרט השיטה הסטטית הפומבית main שחייבת להיות סטטית כי היא נקראת כאשר בכלל אין עצמים בתכנית.
      * אם נרצה לחלק אותה לפונקציות כדי שלא תהיה ארוכה מידי, נהיה חייבים לעשות זאת עם שיטות עזר סטטיות
* **חשוב:**
  + איזה סוגי איברים יכולים ממשקים להכיל?
    - בראש ובראשונה, שיטות ללא מימוש (שבמחלקה המממשת תהיינה פומביות).
    - ממשקים יכולים גם לכלול שיטות ברירת מחדל (default methods), עם מימוש.
    - שיטות ברירת המחדל יכולות לגשת לשיטות פרטיות (עם מימוש).
    - ממשקים יכולים להכיל שיטות סטטיות (עם מימוש).
    - לבסוף, הם יכולים גם לכלול איברים סטטיים סופיים.
* **מחלקת Math:**
  + יש מקרים בהם יש הרבה מתודות עם משהו משותף אבל העשייה שלהן אינה קשורה לאובייקט ספציפי.
  + כלומר היינו רוצים מחלקה שמחזיקה אוסף של מתודות סטטיות, בלי האפשרות לייצר אובייקטים חדשים ממנה.
  + דוגמא לכך היא המחלקה Math או גם המחלקה Arrays

מגבלות של שיטות סטטיות:

* **הצללה:**
  + קונספט בנוגע לפולימורפיזם בשם shadowing, הצללה.
  + בפולימורפיזם ה-reference type הוא זה שקובע לנו מה מותר להריץ וה-object type קובע לנו מה ירוץ.
  + כשאנו מדברים על data members (שדות פומביים או מוגנים) ועל שיטות סטטיות הדבר מתקיים בצורה שונה – רק הרפרנס קובע.
    - לדוגמא אם הגדרנו מחלקה סטטית במחלקה A ומחלקה B שיורשת ממנה תממש גם מחלקה סטטית עם אותה שורת הצהרה נקבל shadowing, לא דריסה של השיטה מ-A.
    - אם נשים אובייקט מטיפוס B ברפרנס של A נסתכל עליו בעיניים של הסוג A –
      * במקרה זה כאשר נריץ את השיטה הסטטית עם שם האובייקט תרוץ השיטה של A, כלומר לפי הרפרנס.
      * גם אם ניגש ל-data member שנדרס נקבל כי הערך שנקבל הוא ערך השדה מ-A
      * כלומר בשני המקרים התלות היחידה היא בסוג הרפרנס.
  + הצללה היא פרקטיקה שאנחנו לא כל כך אוהבים כי זה יוצר קוד יחסית מבלבל.

**6.3 תבנית ה-Facade:**

* **תבנית העיצוב פאסאד:**
  + בא מהמונח מארכיטקטורה של צד של בניין.
  + הרעיון הוא לקחת מערכת מורכבת ולהסתכל עליה דרך חלון צר או דרך אינטרפייס פשוט.
  + פאסאד הוא רלוונטי כאשר יש לנו מערכת גדולה עם הרבה מחלקות ותלויות שונות ביניהן, כלומר יש לנו API מורכב שקשה לעבוד איתו.
  + בהרבה מקרים יש לקוחות שלא צריכים לדעת את כל המורכבות של המחלקה אלא מספיק להם לדעת API פשוט יותר.
  + הפתרון הוא facade – ניקח את המערכת המורכבת ונבנה מחלקה חדשה בשם Facade שתבנה API יותר פשוט כך שהמחלקה עצמה תעשה את העבודה המלוכלכת של להכיר את כל הקשרים ואת הדרך המורכבת לעבוד עם הAPI המקורי, ותספק לנו דרך פשוטה יותר לעבוד איתו.
    - נממש את ה-facade כך שהוא לוקח את המערכת הגדולה ומשתמש בAPI המסובך והפלט שלו יהיה API פשוט.
* **דוגמא:**
  + נניח שיש לנו מערכת מטבח עם מחלקה אבסטרקטית של מוצר ומחלקות שיורשות ממנה, מחלקות של סיר, מחבת, מלחיה.
  + נרצה מחלקה שמכינה פסטה – נבנה מחלקת פסאד שתתמודד עם המורכבות של המטבח ואנחנו נצטרך להתמודד רק עם מחלקה אחת ומתודה שעושה את הפעולה הרצויה.
* **יתרונות השימוש ב-Facade:**
  + הלקוחות רואים רק את ממשק הפאסאד ולא צריכים להתמודד עם הממשק המורכב יותר.
  + אם יש שינויים במערכת המורכבת הלקוחות לא בהכרח מודעים וצריכים להיות מודעים אליהם (כל עוד מחלקת הפאסאד יודעת להתמודד איתם) והלקוחות לא צריכים ללמוד API חדש.
  + אנחנו למעשה לא מאבדים שום דבר – אם הם רוצים לקוחות יכולים להשתמש גם בממשק המקורי.

Facade ופולימורפיזם:

* **פאסאד כמתאם בין טיפוסים:**
  + פאסאד מאפשר לנו לקחת API מורכב ולתת לו חלון ל-API קיים ופשוט יותר.
  + נסתכל על דוגמא של נגני מדיה –
    - יש לנו מחלקה של complexPlayer שיודעת לנגן קבצי מוזיקה עם API מורכב עם המון פרמטרים.
    - יש לנו גם ממשק simplePlayer פשוט יותר שדורש רק קובץ לנגן.
  + ניתן לבנות complexPlayerFacaded שיממש את הממשק של השחקן הפשוט:
    - היא תחזיק אובייקט מסוג complexPlayer כdata member והמתודה שלה תתמודד עם הפרטים המוסבכים של קריאה למתודה המתאימה בcomplexPlayer