**OOP פרק 5 – מנגנוני מחזור קוד**

**5.1 מנגנונים בתכנות מונחה עצמים: שיטות אבסטרקטיות:**

* **מוטיבציה לשיטות אבסטרקטיות:**
  + נניח שיש לנו משפחה של חיות. לכל אחת יש לב, יכולה לנשום וכדומה. אם היינו רוצים לבנות מערכת תוכנה עם היררכיה של מחלקות – מחלקה חיה עם השדות והמתודות המתאימות וכל חיה ספציפית הייתה יורשת ממנה.
  + נניח שהיינו רוצים שכל חיה תוכל להשמיע קול, אבל כל חיה משמיעה כל אחר – אנו רוצים קוד כללי אבל כל חיה תדרוס את הקוד, מה נעשה?
    - נמחק את המתודה של השמעת הקול מהמחלקה של חיה וניצור אותה בכל חיה ספציפית בנפרד – הופך את העיצוב למסובך יותר, מונע שימוש בפולימורפיזם ורעיונית זה שגוי כי למעשה בכל מחלקה המתודה תעשה אותו דבר וזה צריך להיות באותו API
    - מימוש ריק למתודה וכל חיה ספציפית תדרוס אותה ותיתן מימוש אחר – פתרון יותר טוב, אך מה קורה עם מחלקה אחת שוכחת מימוש למתודה? בנוסף, מה המשמעות של קריאה למתודה על חיה כללית?
    - נראה את הפתרון המתאים בחלק הבא
* **מחלקות אבסטרקטיות:**
  + מחלקות אבסטרקטיות הן מחלקות רגילות ששמנו לפניהן את המילה השמורה abstruct.
  + מה שמיוחד במחלקות אלו הוא שלא ניתן ליצור מופע מחלקה (Instance) שלהן .
  + התפקיד שלהן הוא שמהן יצאו מחלקות יורשות.
  + במחלקות אלו ניתן ליצור מתודות אבסטרקטיות – מתודות ללא מימוש.
    - כל מחלקה יורשת ממחלקה אבסטרקטית חייבת לממש את כל המחלקות האבסטרקטיות שיש בה, אחרת נקבל בעיית קומפילציה.
    - כלומר המחלקה האבסטרקטית מכריחה את היורשות לממש את המתודות האבסטרקטיות שלה
  + המחלקה היורשת לא מכילה את המילה השמורה abstruct (גם לא לפני שם מחלקת האב האבסטרקטית)
* **ירושה בין מחלקות אבסטרקטיות:**
  + מחלקה אבסטרקטית יכולה לרשת ממחלקה אבסטרקטית אחרת.
    - במקרה זה היא לא נדרשת לממש את המתודות האבסטרקטיות של מחלקת האב (אבל היא יכולה)
  + מחלקות אבסטרקטיות יכולות להחזיק שדות, מתודות רגילות וקונסטרקטורים – כל דבר שמחלקה רגילה יכולה להגדיר.
    - ההבדל היחיד הוא במילה השמורה abstruct והיכולת להגדיר מתודות אבסטרקטיות.
  + מתי נשתמש בהן?
    - מצב שבו אין הגיון להגדיר אובייקט קונקרטי מהמחלקה הכללית
      * כלומר לאיבר בראש העץ יש משמעות וצורך אבל אין היגיון ליצור מופע ספציפי שלו
    - כאשר אנו רוצים לכפות API על סט של מחלקות

**פולימורפיזם וירושה של ממשקים:**

* **המרה מעלה לטיפוסים גבוהים:**
  + הקשר בין ממשקים לירושה מרובה.
  + ממשקים הם לא חלק מההיררכיה של מחלקות
  + מחלקה יכולה לרשת רק ממחלקה אחת אחרת, אבל מחלקה יכולה לממש כמה ממשקים שהיא רוצה
    - כלומר למחלקה יכול להיות מספר גבוה של types – המחלקה ממנה יורשת וכל אחד מהממשקים שמממשת
  + נשים לב כי גם כל מחלקה יכולה להיות מtype של Object
* מימוש ממשק לעומת ירושה ממחלקה: שניהם מקיימים יחס is/is-a, אבל שימוש מוצדק בירושה מצריך תנאי חזק יותר לפיו לא מספיק ש-A הוא סוג של B, כי אם המאפיין העיקרי של A הוא שהוא B ("A is first and foremost a B"). בהינתן הבחירה בין ירושה ממחלקה למימוש ממשק, נעדיף ממשק. בהמשך נדבר על הסיבות לכך, אבל לפני כן נראה דוגמאות.
* **דוגמאות לממשקים ומחלקה אבסטרקטית מספריית ג'אווה:**
  + ממשקים הם מאוד נפוצים בג'אווה.
  + למשל חבילת הcollections שלה, חבילת מבני הנתונים
    - ממשק Collection – מבנה נתונים כללי
      * מכיל מתודות כמו הוספה, הסרה, גודל
    - ממשק List (יורש מ-collection) – מאפשר גישה לפי אינדקס
      * מכיל מתודות כמו get, set
  + ממשקי Collection –
    - * מדברים יותר על מה ולא על איך
        + למשל יש הרבה דרכים לממש רשימה
      * לכל הממשקים יש אותו API ולכן יותר הגיוני להגדיר אותם בתור ממשקים
  + מחלקה אבסקרקטית – Number שמייצגת מספר כללי
    - יש מחלקות שיורשות ממנה כמו integer, double
    - מכילה מתודות כמו intValue(), floatValue()
* **שיטות ברירת מחדל ופרטיות בממשקים:**
  + עד היום למדנו שממשקים מתפקדים רק בתור חוזה – כל מחלקה שמממשת ממשק צריכה לממש את המתודות שנמצאות בו.
    - כלומר בממשקים יש למעשה שיטות אבסטרקטיות.
  + זאת בניגוד לירושה שבה מחלקת האב שכן מממשת הרבה פעמים את השיטות
  + החל מגרסא 8 של ג'אווה יש אופציה בממשקים לשיטה דיפולטית שמאפשרת לכתוב מימוש לשיטה כבר בממשק עצמו
    - נכתוב default לפני שם השיטה בממשק ואז ניתן להכניס לה גוף.
    - מחלקה שמממשת את הממשק יכולה אבל לא חייבת לממש ולדרוס את השיטה
  + למה צריך שיטות דיפולטיות?
    - עדכונים עתידיים – יכול להיות שכתבנו ממשק, הרבה שיטות מימשו אותו ובהמשך נחליט שצריך להוסיף לו שיטה נוספת. במקרה זה נצטרך להוסיף את השיטה לכל אחת מהמחלקות שמממשות אותו וזה לא תמיד פשוט או אפשרי, ואם הן לא יוסיפו אז הקוד לא יתקמפל.
      * נגיד שהשיטה החדשה האבסטרקטית היא לא תומכת לאחור – backwords compatible
    - שיטות נוחות שמשתמשות בשיטות האבסטרקטיות – אם נניח שהרבה אנשים ישתמשו בשיטות אבסטרקטיות של הממשק באותה צורה נוכל כבר לממש אותה בממשק
    - שיטות דיפולטיביות יכולות להיות שימושיות גם כדי לאפשר לממשקים להיות "ממשקים פונקציונליים" – ממשקים שיש להם שיטה אבסטרקטית אחת בלבד – נדבר על זה כשנדבר על למבדות
  + שיטות עזר פרטיות – ניתן להשתמש בהן רק בתוך המחלקה. במקרה שלנו הן יכולות להיות שיטות עזר לשיטות הדיפולטיות.
  + בשימוש הטהור ביותר של ממשק נרצה להרכיב אותו כמה שיותר משיטות אבסטרקטיות.

אבל יש מקרים שבהם שיטות דיפולטיות ושיטות עזר פרטיות יכולות להיות שימושיות מאוד.

* + לא מומלץ להוסיף יכולות חדשות לגמרי שלא קשורות לשיטות האבסטרקטיות סתם כדי לשתף אותן בין המחלקות – זה שימוש מקביל לירושה
    - מחלקות נועדו למימוש וממשקים נועדו לעשות אבסטרקציה לחוזה.
* **מגבלות של ירושה:**
  + ירושה היא כלי לא עיקרון!
  + חסרונות של שימוש יתר בירושה:
    - שרשראות ירושה ארוכות יכולות ליצור API מנופח ומסורבל במחלקות הסופיות.
    - לאובייקט ממחלקה סופית יהיו הרבה שיטות הפרוסות על מספר מחלקות שונות
    - קוד שקשה להתמצא בו

**5.2 עיצוב תוכנה – הכלה ומחזור קוד:**

* **ירושה לעומת הכלה:**
  + אם כתבנו קטע קוד, מה הדרך הכי טובה לאפשר שימוש בקטע קוד זה במחלקה אחרת?
    - ירושה – מאפשרת מנגנון למחזור קוד. דרך ירושה של מחלקה א ממחלקה ב היא מקבל את כל הAPI של מחלקה ב
    - הכלה – object composition, כלומר הכלה של אובייקט מטיפוס אחר בתוך האובייקט שלנו.
  + האלטרנטיבה לירושה היא הכלה – אנחנו מקבלים את פונקציונלית מחזור הקוד ע"י החזקת אובייקט מסוים שאנו יכולים לקרוא למתודות שלו וכך לקבל את הפונקציונליות.
  + יתרונות של ירושה:
    - ברורה לשימוש, חלק ממנגנון השפה ג'אווה
    - מאפשרת פולימורפיזם
      * לקרוא למחלקה היורשת בשם מחלקת האב
    - מוגדרת באופן סטטי בזמן קומפילציה – לא יכול להשתנות בזמן ריצה
  + יתרונות של הכלה:
    - מוגדר דינמית בזמן ריצה
      * כלומר בזמן ריצה אנחנו יכולים להחליט איזה אובייקט יהיה מוכל
      * מאפשר גמישות בהתאם לנסיבות משתנות
    - מושך לכיוון שכל מחלקה תהיה אחראית על משימה אחת
    - מחלקה יכולה להחזיק כמה אובייקטים שהיא רוצה – לא מוגבלת לירושה אחת
* **דוגמא לשימוש בהכלה על פני ירושה:**
  + מבחינת הAPI בשתי האופציות לא ישתנה
  + ניתן להיפטר מירושות ע"י שימוש בהכלה – תכיל מופע של המחלקה במקום לרשת ממנה.
  + לא צריך ירושה רק כדי לחלק קוד ואחריות בין אובייקטים.
  + עיצוב של הכלה יכול להיות לפעמים ברור יותר
* **מתי נשתמש באיזה מנגנון?**
  + יש שתי מחלקות דומות. האם הן אותו דבר?
  + האם הם סוג של אותו דבר?
    - האם יש להן מימוש משותף?
      * במקרה זה נגיע לירושה
      * האם מחלקת האב מממשת את כל השיטות?
        + אם לא אז אנו צריכים מחלקה אבסטרקטית
        + אם כן אז נבדוק אם ניתן ליצור ממנה מופע נעשה מחלקה רגילה ואם לא אז אבסטרקטית.
        + נשים לב שאם ניתן להשאיר מימוש ריק למתודות מסוימות לא חייבים מחלקה אבסטרקטית
  + אם המחלקות לא אותו דבר – האם הן צריכות לחלוק מימוש או קוד משותף? לחלוק טיפוס?
    - אם כן ניצור קוד משותף או אם צריך שיהיה להן טיפוס משותף ניצור ממשק
  + אם אנחנו לא בטוחים אם הן אותו דבר –
    - אם הן לא חולקות מימוש נעשה ממשק
    - אם הן חולקות מימוש אז נבדוק אם הוא חולק גם מצבים (למשל שדות)
      * אם כן – ממשק והכלה
      * אם לא – ממשק עם מתודות דיפולטיות.
  + העניין של ירושה הוא יחסית נוקשה, פחות גמיש.
    - כאשר נממש יהיה יותר קשה לבצע שינויים
    - ניתן לרשת רק ממחלקה אחת
  + מימוש של ממשקים והכלה יותר גמיש
    - ניתן לממש כמה ממשקים או להכיל כמה דברים שונים
  + האופציה היותר בטוחה לצאת לצד שמאפשר יותר גמישות

**עיצוב תוכנה – אסטרטגיה:**

* **תבנית העיצוב אסטרטגיה:**
  + מצב שבו למערכת שלנו יש משפחה של אלגוריתמים, התנהגויות וכדומה ונרצה לקחת אותם ולהוריד אותם מהמערכת עצמה, כך שיהיה אפשר להחליף בין ההתנהגויות השונות
    - למשל לאפשר לקלט מהמשתמש להחליט באיזו דרך לפעול.
  + הבעיה של אסטרטגיה – נניח שיש לנו כמה וריאנטים לאלגוריתם מסוים (כמה דרכים לבצע משהו, כשבסוף נגיע לאותה תוצאה).
  + הפתרון – תבנית עיצוב אסטרטגיה – נגדיר API (מחלקה אבסטרקטית או ממשק) שמגדיר מה האלגוריתם או ההתנהגות תבצע. לכל אחת מהדרכים לממש את האלגוריתם או לכל אחת מההתנהגויות השונות תהיה מחלקה שתממש את הAPI הזה
    - מפתרון זה נקבל שאת ההתנהגות המורכבת נעביר אותה למחלקה (כך הלקוח והמחלקה הראשית לא מודעים לביצוע)
      * יותר קל להרחבה ושינויים, קוד מודולרי
    - אנחנו עושים הפרדה בין קוד הלקוח לבין קוד ההתנהגות / האלגוריתם. כך בחירת האלגוריתם או ההתנהגות יכולה להיעשות בזמן היצירה או להשתנות בזמן ריצה
      * מקפיץ את השימוש במפעל
  + נניח שנצטרך למיין מערך – יש הרבה דרכים למיין.

נגדיר ממשק אסטרטגיית מיון שיכיל מתודה שמקבלת מערך ותמיין אותו.

נגדיר שתי מחלקות שממשות את הממשק הזה ומממשות את מתודת המיון.

נגדיר מפעל שמכיל מתודה שמחזיר אסטרטגיית מיון

נסתכל על מחלקה שמחזיקה מערך ושדה פרטי מטיפוס אסטרטגיית מיון – נאתחל אותה בעזרת המפעל וניתן יהיה להשתמש בו כדי למיין.

* **אסטרטגיה לעומת ירושה:**
  + אלטרנטיבה לאסטרטגיה היא ירושה - מחלקות שיורשות מהמחלקה הכללית – כלומר להשתמש בירושה כדי לתת פתח למימושים שונים של מיון.
  + למה עדיף אסטרטגיה?
    - אין קיום של התנאי הבסיסי של ירושה is-a
    - היתרון של אסטרטגיה הוא מודולריות - הפרדה בין הלקוח לבין ההתנהגות
    - הסתרת מידע – הלקוח לא צריך לדעת שיש כל כך הרבה אפשרויות או להכריז פומבית בשימוש בהתנהגות או אלגוריתם מסוים.
    - שימוש מחדש בקוד – כל מי שירצה להשתמש במיון מסוים יצטרך לכתוב מחדש את הקוד ולא יוכל להשתמש במה שכתבנו כמו שהיה קורה אם היינו משתמשים באסטרטגיה
    - שינוי בזמן ריצה – נרצה לאפשר למחלקות לשנות התנהגות בזמן ריצה וזה לא אפשרי בירושה

**5.3 ג'אווה סטרים – IDE (המשך):**

* **פתרון בעיה עם ירושה:**
  + נדבר על הדיסקיות במשחק PING:
    - נרצה להשתמש בירושה – ליצור מחלקת אב עם הקוד המשותף של שתי סוגי הדיסקיות.
    - מחלקת האב תהיה מחלקה אבסטרקטית, נדרוס את השיטה update אבל נצטרך ליצור שיטה אבסטרקטית שתחליט את שינוי הכיוון של הדיסקית שהוא שונה לכל מחלקה יורשת ולכן תמומש רק שם.
    - קיבלנו כי אנו עוברים כמה פעמים בין שיטות של מחלקת האב לבין המחלקה היורשת – הרבה קפיצות בין המחלקות בזמן מימוש של שיטה.
      * קשה מאוד לעקוב אחרי זה
    - כמו כן אם יהיו לנו התנהגויות שונות היינו צריכות לייצר עוד שתי מחלקות שיהיו כמו דיסקית המשתמש והAI רק עם שינוי התנהגות מסוים.
  + לכן נעבור לשימוש באסטרטגיה.
  + קיצורי המקשים שלמדנו:
    - ctrl+z כדי לבטל פעולה אחרונה
    - ctrl+y כדי לבטל את ה-ctrl+z האחרון
    - ctrl+end כדי להעביר את הסמן לסוף השורה
* **פתרון בעיה עם אסטרטגיה:**
  + זה לא שהגישות חלופיות, ניתן להשתמש בירושה עד שלב מסוים ואז לעבור לאסטרטגיה.
  + ניצור מחלקה יחידה Paddle שיהיה לה שדה פרטי של משתמש MovementStrategy ובממשק זה תהיה השיטה של חישוב התזוזה
  + ניצור אסטרטגיות תזוזה שממשות את הממשק הזה.
  + כך נוכל לממש את השיטה של חישוב התזוזה בכל אסטרטגיה בנפרד אבל כולם יהיו מטיפוס של האסטרטגיה.
* **Javadoc:**
  + כלי ליצירת תיעוד.
  + יש לו פורמט:
    - תיעוד פונקציה – נכתוב מעליה \*\*/ ונלחץ אנטר ונשלים שם את תיאור הפונקציה ואת ההסברים על הפרמטרים שמקבלת וערך ההחזרה אם יש.
      * בנוסף ניתן להוסיף משתנים של JAVADOC כמו @author או @version
  + יצירת קבצי הHTML –
    - Tools 🡪 generate JavaDoc
    - נבחר אם לייצר עבור כל הפרויקט
    - נבחר את הרמה שבה נחשוף את התיעוד – פומבי, מוגן או פרטי
    - נגדיר איזה משתנים לכלול (@param, @return, @author וכו) – רק אותם נראה בקובץ HTML גם אם אחרים מוגדרים
  + תיאור חבילה – אם נלחץ מקש ימני על שם של חבילה ואז

New 🡪 package-info.java זה יוסיף ספציפית עבור JAVADOC את תיאור החבילה – כלומר בקובץ הHTML יופיע תיאור של החבילה.