**תרגול 12 – OOP**

**תכנות פונקציונלי:**

* פרדיגמת תכנות בה התוכנית נבנית ע"י הפעלת פונקציות
* מבוססת על lambda calculus
* אנחנו מתייחסים לפונקציות בתור first class citizens – אפשר להעביר לפונקציות, לשמור בשם וכל מה שאפשר לעשות עם כל טיפוס אחר.
* ממשקים פונקציונלים יכולים לאחסן ביטויי למבדא או רפרנסים למתודות
* לקונסטרקטור נפנה ע"י ClassName::new
* כל ממשק עם SAM (single abstract method) הוא ממשק פונקציונלי
  + מתודות דיפולטיות לא אבסטרקטיות ולכן לא נחשבות
  + לממשק פונקציונלי יכולות להיות כמה פונקציות דיפולטיביות
  + מומלץ להשתמש ב@FunctionalInterface
  + יש לנו בjava.util.function ממשקים פונקציונליים, לדוגמא:
    - Function<T,R> - קלט יחיד ופלט יחיד, לרוב הטיפוסים הפרימיטיביים יש ממשקים מוגדרים מראש לפי טיפוס
    - BiFunction<T,U,R> - מקבל שני ארגומנטים ומחזיר אחד, גם פה יש לפרימיטיביים ממשקים מוגדרים מראש
    - Supplier<T> לא מקבל ארגומנטים ומחזיר פלט
    - Consumer<T> - מקבלת ארגומנט ולא מחזירה כלום
    - Predicate<T> – מקבלת ערך ומחזירה בוליאני
  + אם אין משהו שמתאים לנו תמיד אפשר ליצור משלנו

**שלדי תוכנה – Framework:**

* אוסף של קוד שנכתב מראש המשתמשות כתבנית שניתן להשתמש בה ע"י מילוי בהתאמה לקוד שאנו צריכים
* הן מספקות ממש שלד לתוכנית
* נשתמש בו ע"י קריאה למתודות מתוכו או ע"י ירוששה, סיפוק callbacks או listeners או כל מימוש של Observer pattern (משהו מחזיק כמה observers שכאשר מתבצע שינוי כל אחד מהם משנה במקום המתאים)
* Frameworks vs libraries:
  + ספרייה היא אוסף של מחלקות נפוצות שמקומפלות מראש, לעומדת FRAMEWORK שיהיה משהו יותר גדול שיכיל כמה ספריות
  + ספרייה היא אוסף פונקציות שאנו יכולים לקרוא להן מתוך מה שאנו כותבים ואילו FRAMEWORK ממש מגדיר לנו את המבנה שבו אנחנו כותבים – ארכיטקטורה וכו
  + בשימוש בספריות השליטה היא אצלנו, לעומת FRAMEWORK – אף פעם לא נקרא לFRAMEWORK עצמו, הוא זה שיהיה אחראי לקרוא לדברים שלנו, למתודות שאנו מספקים לו (שיכולות לקרוא למתודות עזר אבל בכללי הוא קורא למה שאנחנו נכתוב)
  + קל לשלב ספריות בתוכנית שלנו, ואילו FRAMEWORK זה לא משהו שניתן לשלב אלא אנו צריכים להתחיל ממנו ולהתאים אותו למה שנרצה לכתוב.
  + החשיבות העיקרית של הספריות היא אחסון קוד שיעזור לנו בLINKING ובתהליך הBINDING ואילו FRAMEWORKS מספקים דרך סטנדרטית לבנות דברים
  + עיקרון הוליווד \*\*\*
* Inversion of Control – היפוך שליטה:
  + עיקרון בהנדסת תוכנה שאומר שאנו מעבירים את השליטה על האובייקטים או על חלקים מהקוד לקונטיינר או לFRAMEWORK
  + שונה מתכנות מסורתי כי מאפשר לFRAMEWORK לקחת את השליטה על ריצת התוכנה
  + ניתן לממש בכמה דרכים שונות – למשל אסטרטגיה (האסטרטגיה מתקבלת מבחוץ) או מפעל
  + יתרונות עיקריים:
    - מנתק את הביצוע של המשימה מהמימוש שלה
    - קל יותר להחליף בין מימושים שונים
    - מאפשר מודולריות לתכנית
    - קל יותר לבדוק דברים בכך שנבודד התנהגות של חלק מסוים

**Dependency Injection:**

* אחת הדרכים לממש עיקרון זה
* תבנית עיצוב שנועדה לממש עיקרון זה. מאפשרת יצירה של אובייקטים תלויים מחוץ למחלקה ולספק את האובייקטים עצמם למחלקה בדרכים שונות.
* בשימוש בDI אנו מזיזים את יצירת וקשירת האובייקטים התלויים למחוץ למחלקה שתלויה בהן.
* כוללת 3 סוגי מחלקות:
  + המחלקה התלויה – CLIENT CLASS – מחלקה שתלויה במחלקה שמספקת לה שירות
  + SERVICE CLASS – מספקת שירות למחלקת הלקוח
  + INJECTOR CLASS – מספקת למחלקת את הלקוח את אובייקט מחלקת השירות שהיא צריכה
* מפרידה את האחריות ליצירת אובייקט של מחלקת השירות למחוץ למחלקת הלקוח
* למעשה הINJECTOR מייצר אובייקט אחד של מחלקת השירות ומספק אותו למי שצריך ממחלקת הלקוח.
* הInjector בדר"כ מזריק את התלויות ב3 דרכים:
  + Construction Injector – הINJECTOR מספק את השירות (התלות) דרך הקונסטרקטור של אובייקט הclient class
  + Property / Setter Injection – הINJECTOR מספק את התלות דרך public property של מחלקת הלקוח (כמו פונקציית setter)
  + Method Injection – הלקוח יממש ממשק שמצהיר על מתודה/ות שמספקות את התלות, והINJECTOR משתמש בממשק זה כדי לספק את התלות למחלקת הלקוח.
    - כמו מעטפת לדרך השנייה
    - בפועל הלקוח שיממש את המחלקה הזאת יהיה השמה של משהו לשדה ספציפי
* הרבה FRAMEWORKS מאפשרים יצירה אוטומטית וניהול של כל התלויות באובייקטים בשימוש בannotations וקבצי XML