**פרק 8 – OOP**

**למבדות:**

**מחלקות מקומיות ואנונימיות:**

* למבדא זה קטע קוד (שורה שתיים) שניתן להתייחס אליו כמו אל מידע – לשים במשתנה לשלוח לפונקציה וכדומה
* איטרבל זה הממשק של JAVA שאפשר לעשות עליו איטרציות
* פרמטריזציה – לקחת משהו ולהכניס אותו לפרמטר
* מספר טיפוסים בקובץ – מותר לשים כמה טיפוסים באותו קובץ כל עוד רק אחד מהם הוא פומבי
* WRAPPER – מחלקה עוטפת
  + Boxing – לקחת int וליצור ממנו integer
  + Unboxing – לקחת ממחלקה הWRAPPER ולהפוך לפרימיטיב
* איך לשים דברים שקשורים ביחד בתוך הMAIN
* מחלקה שמופיעה בתוך שיטה היא מחלקה מקומית local class –
  + אפשר לראות אותה בכל מקום החל מרגע הגדרתה וכאשר יוצאים מהשיטה לא נוכל לראות אותה
* ברוב המקרים שאנו צריכים מחלקה באופן כל כך לוקלי אנחנו בעצם צריכים זה רפרנס לאובייקט שלה וJAVA מציעה לנו פתרון ספציפי יותר – מחלקה אנונימית –
  + מחלקה שאפילו לא נציין את השם שלה
  + נניח שיש לנו ממשק A, נגדיר מופע של מחלקה המממשת את הממשק אך עם שם הממשק (כלומר לא ניתן שם למחלקה). היא תעבוד על בנאי ריק ואחרי ההצהרה נפתח סוגריים מסולסלים ונכתוב בפנים את השיטה
    - למעשה אנו עושים upcast
  + מחלקות אנונימיות תמיד עובדות עם בנאי ריק
* מחלקה אנונימית יותר נפוצה ממחלקה לוקלית

**למבדות וממשקים פונקציונליים:**

* למבדות:
  + למבדה לוקחת תחביר של מחלקה אנונימית, זורקת ממנו כל מה שJAVAיודעת להבין לבד ונשארת רק עם מה שנחוץ
  + אנו יכולים למחקו את שורת ההצהרה של הפונקציה כי JAVA יודעת שמופע המחלקה שממשת ממשק חייבת לממש את הפונקציה הספציפית.
  + נישאר עם שם המשתנה שנשלח לפונקציה, חץ ימינה ונכתוב את מה שנראה לעשות בפונקציה (למשל במקרה שלנו הביטוי – אם המספר מתחלק בשתיים)
  + נקבל משהו שנראה כמו פונקציה, ושם הממשק ישתנה לממשק פונציונלי
  + מבנה של למבדה: <parameters -> \_\_\_\_;
    - אם השיטה לא מקבלת כלום נשים במקום הפרמטרים לפני החץ סוגריים ריקים ()
* ממשקים פונקציונליים:
  + התנאים של ממשק שניתן לממש ע"י ביטוי למבדה –
    - ממשקים פונקציונליים לא ממש מייצגים מחלקה אלא מייצגים פונקציה
    - יש להם רק שיטה אחת שצריך לממש (אם יהיו יותר לא נוכל לממש עם למבדה)
    - כן יכולה להכיל עוד פונקציות אם לא צריך לממש אותן – שיטות פרטיות, שיטות דיפולטיביות
  + יש לנו תגית @FunctionalInterface שאומרת שזהו ממשק פונקציונלי
    - ברגע שנוסיף את התגית לא נוכל להוסיף לו עוד פונקציות יותר מאחת שצריך לממש
* Java.util.function:
  + JAVA מספקת לנו ממשקים פונקציונלים נפוצים בהם היינו יכולים לרצות להשתמש
  + Java.util.function.predicate – ממשק המייצג פונקציה שמקבלת טיפוס T ומחזירה בוליאני אם עבר או לא עבר תנאי כלשהו
    - נצטרך להוסיף בהצהרה Predicate<T>
  + יש המון ממשקים קטנטנים שיכולים לחסוך
  + ממשק נוסף Consumer – ממשק שמייצג א הפונקציה המקבלת טיפוס T ובולעת אותו – לא מחזירה כלום.
    - שם הפונקציה היא accept
    - הפונקציה עושה איתו משהו אך מחזירה void
  + ממשק Supplier – לא מקבל פרמטר אך מחזיר משהו
    - פונקציה שמספקת באופן קבוע איזושהי לוגיקה
  + הממשק Function – מייצג פונקציה שמקבל T ומחזירה R (שניהם משתנים גנריים)
    - Function<T,R>

**רפרנסים לשיטות:**

* Method References:
  + אפשר ליצור למבדה גם עם בלוק ולא רק עם שורה – אחרי החץ נפתח בלוק עם סוגריים מסולסלים
  + מחלקות קטנות מסרבלות את העניינים
  + אם אנו רוצים לשלוח שיטה A שנמצאת במחלקה נכתוב בפרמטר שאנו שולחים Lambdas::A
    - :: אומר לנו שאנו לא שולחים קריאה לפונקציה אלא שולחים את הפונקציה עצמה
    - שיטה סטטית – נגיד שאיזו מחלקה היא נמצאת
  + אם למשל נרצה לשלוח פונקציה הדפסה – System.out::println
    - זה יהיה שיטת מופע של האובייקט OUT – ולכן כתבנו את האובייקט עם נקודותיים נקודותיים אחרי ואז שם הפונקציה
  + כלומר עבור שיטה סטטית נגדיר באיזו מחלקה נמצאת ועבור שיטת מופע נגיד באיזה אובייקט היא נמצאת
  + אלו למעשה רפרנסים לשיטה – זו דרך יותר מפורשת להגיד מה אנו רוצים
  + רפרנסים לשיטות מאפשרים לנו להשתמש עבור ממשקים פונקציונליים בפוקנציות קיימות במקום לעשות רפרנס כפול (שיטה שקוראת לשיטה אחרת.

**משתנים קבועים אפקטיבית:**

* Effectively Finals:
  + JAVA מאפשרת להשתמש בתוך למבדות, מחלקות אנונימיות ומחלקות לוקליות במשתנים שהמחלקה החיצונית מכירה – גם משתנים שקיבלה וגם משתנים היא הגדירה.
  + כלומר JAVA תומכת בסוג של מה שנקרא בשפות אחרות closure
  + המשתנה שביטוי למבדה יכול להשתמש בו אמור להיות final או effectively final – משתנה שמוכר ע"י השיטה החיצונית ולכן ניתן להשתמש בו בלמבדה נדרש שיקבל ערך אחד ולא משתנה יותר
    - לא חייב שיהיה כתוב עליו את המילה final, העיקר שיתנהג כמו final
    - ברגע שמשתנה הוא כבר לא קבוע אפקטיבית לא נוכל להשתמש בו בלמבדה
  + המשתנים נדרשים להיות קבועים אפקטיביים אך לא immutable – כלומר נוכל למשל להשתמש במשתנה שהוא רפרנס, לא לשנות את הרפרנס עצמו אך את מה שיש בתוך הרפרנס כן נוכל לשנות

**Callbacks:**

* קולבאקס – הכוונה היא לפרמטר שהטיפוס שלו הוא פונקציה
  + כל פונקציה שמישהו אחר שלח לנו כדי שאנחנו נקרא לה.
* ממומשות באמצעות ממשקים פונקציונליים
* הממשק Runnable – ממשק פונקציונלי מובנה בJAVA ויש לו שיטה run שלא מקבלת כלום ולא מחזירה כלום
  + כלומר מייצג פונקציה שלא מקבלת כלום ולא מחזירה כלום
* ניתן לשלוח כcallback למבדה או רפרנס לשיטה
* מאפשרות להרחיב את השירות של API קיים וגם מאפשרים להגדיר מחלקות שצורת החשיבה סביבן היא אחרת

**Callback Strategies:**

* ירושה לעומת אסטרטגיה – בחינה מחודשת:
  + פעם ירושה הייתה פתרון שקל יותר לממש ואסטרטגיה נתנה לנו גמישות בתמורה למורכבות או סרבול
  + בעיה – נרצה לממש כדור ולבנה עם נקודת המוצא של GAMEOBJECT
    - מופע של מחלקה כזו נותן לנו לייצג אובייקט בגודל כרצוני, עם ציור כרצוני ועם מהירות כרצוני.
  + פתרון 1 – ירושה:
    - המחלקה מגישה שיטה עבור התנהגות
    - המחלקה של הכדור תירש מGAMEOBJECT ונדרוש פונקציה זו, ובדומה עבור הלבנה
    - נקבל מחלקה של GAMEOBJECT וממנו ירשו מחלקות הכדור והלבנה ואם נרצה לבנים עם התנהגות מיוחדת הן יהיו מחלקות שיורשות ממחלקת הלבנה
  + פתרון 2 – אסטרטגיה:
    - בגישה זו נוכל להסתכל על הכדור והלבנה כסוג אחד של עצם פשוט עם התנהגות שונה במקרה של התנגשות.
    - נגדיר מחלקה של התנגשות אובייקט שיורשת מGAMEOBJECT, היא תכיל שדה של אסטרטגיית התנגשות COLLISIONSTRATEGY.
    - בשיטה שנדרסת נקרא גם למימוש האב ואם הגדרנו אסטרטגיית התנגשות נפעיל אותה
    - כך יספיק לנו מחלקה אחת – גם הכדור וגם הלבנה יהיו מופעי מחלקה של המחלקה הזו, אולם כן נצטרך להגדיר ממשק COLLISIONSTRATEGY, ולאחר מכן אסטרטגיית התנגשות לכדור ואחת אחרת ללבנה (מחלקות שמממשות את הממשק)
    - כלומר יהיו לנו את GAMEOBJECT ומחלקה שיורשת ממנו, את הממשק ושני מחלקות שממשות אותו לכל התנגשות, ואם נרצה התנהגויות נוספות ללבנים הן ירשו מהמחלקה שמממשת את ההתנהגות של הלבנה
* Callback Strategies:
  + פתרון 3 – מימוש אסטרטגיה באמצעות CALLBACK
  + במקום להגיד שנקבל אובייקט מחלקה של COLIISION STRATEGY נקבל CALLBACK
  + הCALLBACK צריכה לקבל את העצם שמתנגשים בו, מידע על ההתנגשות ולא להחזיר כלום
    - ממשק פונקציונלי BiConsumer מייצג פונקציה שמקבלת שני פרמטרים ולא מחזירה כלום
    - השיטה נקראת accept
  + שאר הפתרון חוץ מהפרמטר שנקבל יראה אותו דבר כמו בפתרון האסטרטגיה.
  + במקום לחשוב על האסטרטגיה כמופע מחלקה נחשוב עליה כפונקציה
  + כמו כן, אנחנו לא צריכים לממש את הממשק הפונקציונלי ולא להגדיר מחלקות – נשתמש בלמבדות או רפרנסים לשיטות.
  + נשים לב כי אנו יכולים להשתמש במשתנה שעבורו אנו שולחים למבדה בתוך הלמבדה עצמה
  + בפתרון זה נצטרך ליצור רק מחלקה אחת נוספת לGAMEOBJECT
  + איך נגדיר אסטרטגיה חדשה?
    - נשלח למבדה אחרת או רפרנס לשיטה אחרת
  + תמיד אפשר לממש CALLBACK גם ע"י מחלקה שמממשת את הממשק הפונקציונלי, אך אפשר גם עם ביטוי למבדה או רפרנס לשיטה
  + היתרונות של Callback Strategy:
    - שינוי האסטרטגיה בזמן ריצה
    - גמישות – לא מוגבל ע"י עץ ירושה
    - ריבוי אסטרטגיות – ניתן להגדיר כמה CALLBACKS לאותו GAMEOBJECT, כאשר כל אחת תטפל בסוגיה אחרת
    - קוד קצר וברור
    - מודולריות – נשלוט בנפרד באסטרטגיה ובנפרד בGAMEOBJECT
    - Code reuse – ניתן לשלוח את אותו CALLBACK לעצמים מסוגים שונים
  + כל מה ששינינו במחלקה שיורשת מGAMEOBJECT הוא רק שורות קטנות – מי שכתב את GAMEOBJECT יכול לחשוב על זה ועם כמה שינויים קטנים היינו יכולים לשנות את GAMEOBJECT ולא נצטרך להשתמש בטיפוסים שונים בכלל.

**משנה זהירות:**

* CALLBACKS מאפשרות למחלקה אחת לתת מענה בקלות להרבה צרכים
* כמו כן CALLBACKS מאפשרות לממש אסטרטגיות בלי טיפוסים נוספים, בלי ממשק האסטרטגיה והמחלקות שמממשות אותו.
* CALLBACKS הן כמו דרך חינמית להרחיב את הAPI.
* לכן נרצה להיזכר ב-Minimal API – ככל שהשירות מצומצם יותר הוא:
  + יותר קל להבנה
  + תפקיד המחלקה מוגדר
  + תקינות – יש פחות באגים
  + קל יותר לתחזק
  + קל יותר לבדוק – TESTABILITY
* לא עומת זאת – הוא פחות נוח, יותר קשה ללקוח
* CALLBACKS יכולות לקחת אותנו מהכיוון המינימלי לכיוון הנוח צריך לשים לב לtradeof
* אבל זה קצת יותר מורכב – ככל שמוסיפים יותר אסטרטגיות/callbacks למחלקה, עבור מי שיוצר את המופעים עצמם זו לא באמת "בחינם" – הייצור של עצם אחד יכול לקחת הרבה שורות קוד עבור קוסטומיזציה באמצעות למבדות שהם CALLBACKS ואסטרטגיות
* הייצור יישב במחלקה ראשית שיכולה לעשות גם עשרות אתחולים של מופעים שונים.
* תחומי אחריות –
  + איזו מחלקה לוקחת אחריות מוגדרת היטב על הקוד של אתחול - כדי לשמור על הסדר הטוב ועל אנקפסולציה נפריד את האתחול לקובץ משלו
    - ניצור מחלקה שאין לה קשר ממשי שאין לה קשר ממשי לאובייקט שאנו יוצרים בתוכה חוץ מזה שמייצגת את האתחול שלו עם שיטה סטטית שיוצרת אובייקט
    - יכול להיות שלעצמים אחרים נגדיר מחלקות אחרות
  + נקבל יחידה מוגדרת היטב שאחראית על סוג מסוים של אובייקטים עם אסטרטגיות וCALLBACKS ספציפיות וזה מתחיל להיות דומה לקפסולה – תחום אחריות מוגדר, מימוש מוסתר אבל לא – זו שיטה סטטית.