שיעור 1

1.1

ארבעת המושגים הבסיסיים ב-OOP הינם:

אובייקטים: אבני הבניין של העולם הווירטואלי, המייצגים ישויות בעלות תכונות והתנהגויות.

מחלקה: שרטוט או תבנית ליצירת אובייקטים. אובייקטים מאותו סוג שייכים לאותה מחלקה, חולקים תכונות והתנהגויות משותפות.

תכונות (attributes): מאפיינים או מאפיינים הקשורים למחלקה, המגדירים את התכונות של האובייקטים שלה.

שיטות:(methods) פעולות או התנהגויות שאובייקטים של מחלקה יכולים לבצע, המוגדרים בתוך המחלקה.

לסיכום, כל עצם הוא מופע של מחלקה כלשהי. המחלקה מגדירה אילו תכונות ושיטות יהיו לכל העצמים שלה.

הריצה של תוכנה מונחית עצמים היא האינטראקציה שבין העצמים.

כל מחלקות ה-java יושבות בקבצים שנקראים ClassName.java

Data-members זה התכונות שמאפיינות את המחלקה הזאת.

בנאי

- הבנאי יכול לקבל כל סוג פרמטר כמו שיטה רגילה.
- בנאי הוא שיטה מיוחדת שנקראת כשיוצרים עצם.
- ישנו בנאי דיפולטיבי שמוגדר עם יצירת מחלקה ונדרס עם יצירת בנאי אחר.

כל משתנה שאנחנו מגדירים בג'אווה יכול להיות אחד מבין שני הדברים הבאים, הוא יכול להיות הפניה לאובייקט מסוים, כלומר הוא יכול לייצג איזשהו אובייקט, לחלופין הוא יכול להיות משתנה פרימיטיבי.

מה זה אומר משתנה פרימיטיבי?

זה אומר משתנים שהם דברים פשוטים כמו מספרים ואותיות שהם לא מחלקות בג'אווה.
פרימיטיביים בג'אווה מחזיקים את סוגי המידע הכי בסיסיים שיש.

מחרוזת או String באנגלית מיוצגת על ידי מחלקה בג'אווה

מחרוזות בג'אווה הן עצמים! לכן x ו y-הם לא המחרוזות עצמן, אלא רק מצביעים על המחרוזות. השאלה x==y פשוט משווה בין המצביעים. ג'אווה עשויה להשיב false על השאלה x==y ועשויה גם להשיב true כתלות בהאם מאחורי הקלעים שני המצביעים מצביעים על אותו עצם בזיכרון. על מנת לבדוק האם שתי המחרוזות שקולות, גם אם הן שמורות בשני עצמים נפרדים, נשתמש בשיטה equals של String: x.equals(y). רפרנס זה לא אובייקט אמיתי אלא איזשהו חץ שמבציע לאובייקט קונקרטי. לכל רפרנס יש את הטיפוס שלו שזה השם של המחלקה.

ה Garbage Collector-זה איזשהו יצור עלום שם שמגיע ומשחרר זיכרון של אובייקט שאין רפרנס אליו. כלומר הוא מחזיר את הזכרון לרשות התוכנה כדי שהיא תוכל לעשות בו שינויים, שימוש בזמן עתיד.

null הוא כלום. היעדרות של אובייקט .כאשר אנו מגדירים מצביע לאובייקט, הוא יכול להצביע לאובייקט ממשי קיים או לא להצביע לכלום .למשל, כאשר אנו מגדירים Bike example = null; לא נוצר עצם כלל והמצביע example מצביע על כלום.

מחלקת ה-String ב-Java היא מחלקה בסיסית ונפוצה בשימוש. היא דומה לפרימיטיביים אבל לגמרי מחלקה. ניתן להגדיר רפרנס באמצעות הסימן '=' ללא בנאי. המחרוזות אינן ניתנות לשינוי, כלומר לא ניתן לשנות את התוכן שלהן לאחר שנוצרו. סטרינג מציעה שיטות שימושיות שונות, כגון length ו-charAt. אי-שינוי מבטיח שברגע שמחרוזת נוצרת, התוכן שלה נשאר קבוע.

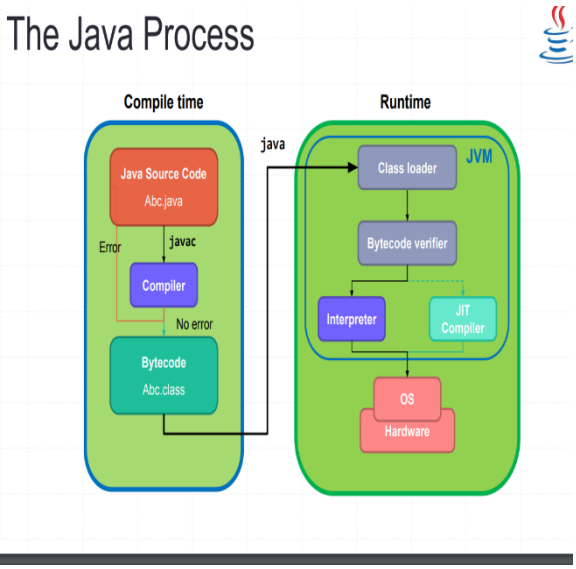
Mainהיא פונקציית Java הראשית שדרכה פועלות בדרך כלל תוכניות, בדרך כלל התוכנית מסתיימת כשהסתיימה הפוקנציה.main Mainהיא פונקציה או שיטה מיוחדת עבור.Java

אתחול מערך: int[] intArray = new int[]{1, 2, 3};

למערך קיים משתנה בשם length

ב-Java, כשאנו כותבים קוד, ישנן קונבנציות וכללים שיש להם ערך גדול:

- אינדנטציה\הזחות: שימוש נכון בהזחות ביחס לסוגריים. זה משפר את הקריאות והסדר.
- סימון של סוגריים: סוגריים מסולסלים ב-Java מתחילים באותה שורה כמו התחלה של מה שפתח אותם.
- Camel Case Notation: שמות מחלקות מתחילים באות גדולה, ובשם של תת-תכנית, שדות, ושיטות מתחילים באות קטנה.
- שימוש בקבועים גדולים: שמות של קבועים מוגדרים באותיות גדולות.



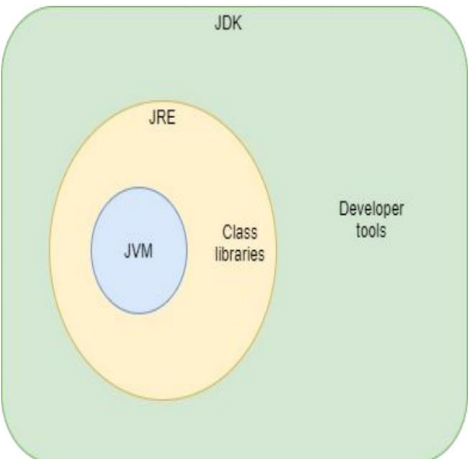
JDK / SDK

JDK – ערכת פיתוח Java.

- ה-JDK הוא ערכת התכנות המלאה שמפתח צריך לתכנת ב-Java. זה כולל תוכניות כגון
  - מהדר (javac), ה-
  - מכונה וירטואלית Java Virtual Machine (javac)
  - diassembler, (javap)
  - מאתר באגים (debugger)
  - כלי תיעוד (javadoc).
- חשוב לציין: JVM ו-JRE כלולים ב-JDK.

▪ SDK - Software Development Kit –מתייחסת לכל חבילת תוכניות שמתכנת צריך כדי לכתוב קוד. אז JDK הוא Java SDK.

JRE



- JRE – סביבת זמן ריצה של Java.
- קבוצת משנה של תוכניות וספריות הכלולה ב-JDK. זמן הריצה של Java סביבה מתייחסת לכל הכלים והתוכניות המסייעים בביצוע תוכנית Java. זה כולל את הjvm וסיפריות של מחלקות שונות בjava.

:Compile Timse

קוד המקור של Java מורכב באמצעות הפקודה 'javac'. מהדר מייצר Bytecode עם סיומת.class.
:Bytecode

Bytecode הוא קבוצת פקודות שאינה ניתנת לקריאה על ידי אדם.
כל פעולת קוד בתים היא באורך ביית אחד.
ניתן להשתמש ב-'javap -c' כדי לבדוק קוד בתים בצורה קריאה אנושית.
ביצוע זמן ריצה:

הפקודה 'java' קוראת לכל הכלים הדרושים לנו לזמן ריצה, היא גם מפעילה את ה-JVM.
- אנו מאכילים את ה- JVM ב bytecode ישירות.
- ה-JVM טוען תחילה את ה-bytecode שלנו וחלק מקוד הספרייה של Java (Java.utils וכו').
- הclass loader יוצר זיכרון כללי לכל מחלקה
- ה byte code verifier מוודא שקוד הבתים תקף (פורמט תקין ונעשה על ידי מהדר מהימן)

מנוע ביצוע=Execution Engine : מורכב משלושה פריטים (תלוי ב-JVM)
1. interpreter - מתרגם bytecode לקוד מכונה שורה אחר שורה לביצוע מיידי
2. JIT – Just In Time Compilation, אם JVM רואה גוש של קוד חוזר ונעשה בו שימוש שוב ושוב, זה יכול לקמפל את הנתח הזה (במהלך זמן הריצה) כדי לייצר קוד אופטימלי יותר, מכיוון שפירוש שורה אחר שורה יכול להיות לא יעיל.
3. garbage collector
- חלק ממנוע הביצוע, לכל jvm יש Garbage Collection שתפקידו "לנקות" את הזכרון.
- מנקה אוטומטית זיכרון תפוס על ידי אובייקטים ללא הפניות.
- ניתן להשוות למצביעים משותפים ב-C++ אך עם שליטה אוטומטית.
- G-C פועל כאשר ה-JVM רואה בכך צורך, ולא ניתן לשלוט בו ישירות.
יתרונות וחסרונות בJAVA

Pros	Cons
▪ **WORA** (Write Once Run Anywhere) Bytecode works universally for all JVMs. The JVM is system specific, so it takes care of the JVM. ▪ Java is backwards compatible, we can compile using JDK8 and run it on a JVM in JDK16.	▪ Interpreting and compiling using JIT costs us in runtime performance. ▪ Automatic memory management leads to performance issues

<u>אנקפסולציה</u> - אנקפסולציה אומרת בגדול לחלק את הקוד לקפסולות, והיא מורכבת משני חלקים.

החלק הראשון הוא קודם כול לאגד ליחידה אחת את כל הפונקציות וגם את כל המשתנים שעוסקים באותו תחום אחריות.

החלק השני של אנקפסולציה *הסתרת מידע* הוא שכל עצם מפריד בין מה שמשרת יחידות אחרות ומה שנועד לשימוש פנימי. חוץ מכמה פונקציות, כל עצם מסתיר מהעצמים האחרים את הפונקציות ואת המשתנים שלו כאילו הם לא קיימים.

<u>אז אנקפסולציה אומרת:</u> שאנחנו קודם כול מחלקים את הפונקציונליות בתוכנה לקפסולות שמרכזות תחומי אחריות, ובנוסף הקפסולות מסתירות כמה שיותר איברים. **כל עצם נולד כדי לוודא תפקיד מוגדר!**

יתרונות האנקפסולציה:

- קל לתת היגיון לקוד שכן אובייקטים אחראים לדבר אחד
- לדעת מי אחראי -> לדעת היכן לחפש באגים מסוימים.

אנקפסולציה עשויה לעזור במקרים הבאים:

- ניפוי שגיאות (דיבוג) - מכיוון שהקוד מחולק לחלקים לוגים, אפשר למקד בדיקות לאיתור תקלות.
- רכיב שעומד בפני עצמו ניתן לשלב גם בתוכנות אחרות.
- חלוקת עבודה בין חברי הצוות.
- שינוי של המימוש במחלקה אחת לא מצריך שינוי במחלקות אחרות.

<u>הסתרת מידע-כימוס</u> מאלצת אותנו לכתוב קוד שקשור לתחום האחריות של אובייקט בתוך האובייקט.

אחת המטרות העיקריות של הכימוס היא הפרדה בין פונקציות ומשתנים הנועדו לשימוש פנימי בתוך המחלקה לבין אלו הנועדו לשימוש חיצוני מחוץ למחלקה.

עקרון הכמוס מנחה אותנו להסתיר מידע על המימוש.

<u>API</u> ראשי תיבות של.Application Programming Interface והדגש כאן הוא על Programming Interface, כלומר הממשק התכנותי.=הממשק של העולם החיצון עם המחלקה.

כשאנחנו ניגשים לכתוב קוד חדש, נעבוד בשלושה שלבים.

שלב ראשון – אנקפסולציה, כלומר נחלק את העצמים לקפסולות לפי תחומי אחריות.

השלב השני הוא לקבוע מה יהיה הממשק, ה,API- של כל עצם. זה החוזה של המחלקה עם הלקוחות שלה.

רק השלב השלישי הוא המימוש של המחלקה. עכשיו, מאוד יכול להיות שבשלב השלישי, כשנבוא לממש את החוזה הזה, נצטרך שדות או נחלק לשיטות או נוסיף קבועים, אבל כל האיברים שאנחנו מוסיפים בשלב המימוש הם פרטיים. אז השאלה איזה איברים צריכים להיות פרטיים היא קלה. כל מה שמתווסף בשלב המימוש הוא פרטי.

השאלה היותר מעניינת היא איזה איברים מגדירים את החוזה.

עקרון הממשק המינימלי,**Minimal API** , הוא עיקרון שאומר דבר פשוט: Keep It Simple.

האובייקט צריך לעשות כל מה שמצופה ממנו, אבל הוא גם צריך לעשות רק מה שמצופה ממנו, וכל המוסיף גורע. למה גורע?

מנוקדת המבט של הלקוח, יותר פונקציונליות זה יותר להבין, יותר לזכור ויותר מקום לטעויות.

מנקודת מבטח של המתכנת, לקיחת אחריות נוספת שוברת אנקפסולציה.

אבסטרקציה היא התרחקות מהמימוש והתקרבות לצורך של הלקוח. הרעיון הינו להסתיר מידע מיותר בקוד שלנו ולהראות רק את מה שחיוני.

הפשטה טובה לא מתייחסת לפרטים של "איך" משהו נעשה ורק אומרת לנו "מה" נעשה.

יתרונות האבסטרקציה:

- גורם לקוד שלנו להיות קל יותר להבנה ולשימוש.
- מנתק את היישום שלנו מהשימוש בקוד. (בהמשך שינוי היישום לא אמור להשפיע על המשתמשים).

לאבסטרקציה של ה API-יש עוד יתרון גדול דווקא מהצד שלנו, המפתחים.

התחלנו מאנקפסולציה ואמרנו שהיא מחלקת את הפונקציונליות לתפקידים. ובתהליך הזה גם קיבלנו את "מה" שהעצם צריך לעשות, וזה החלק הקבוע. "איך" שהוא עושה את זה, זה תמיד נתון לשינוי.

כל עוד ה API-שלנו אבסטרקטי, כלומר מורחק מהמימוש, הוא לא משתנה אפילו אם המימוש משתנה. ואם ה API-לא משתנה, אז השינויים לא מתפשטים, ואז קל יותר לעשות טסטינג, חלוקת עבודה, דיבוג, החיים נהיים טובים.

בואו נסכם את כל מה שאמרנו על ה.API-

דיברנו על שני עקרונות שעוזרים לנו להנחות את ההגדרה של ה API-של מחלקה.

ממשק מינימלי ואבסטרקציה.

לסיכום בניים ותזכורת:

נסכם את שלבי הפיתוח של תכנית מונחית עצמים קטנה, ואלו הם:
1 אנקפסולציה.
לחלק את התכנית למחלקות, כך שכל מחלקה אחראית על נושא אחד מוגדר היטב.
2 <u>קביעת הAPI</u>
אמרנו ש API-הוא הממשק התכנותי של המחלקה . כאן אנחנו מבטאים את התפקיד של המחלקה בבחירת האיברים פומביים שלה. דיברנו גם על שני עקרונות שעוזרים לנו בשלב קביעת ה:API-
2.1 מינימליות API - לא להכביד על הממשק. להעדיף רשימת שיטות פשוטה להבנה ופיענוח.
2.2 אבסטרקציה :קודם לחשוב איך היינו רוצים שהמחלקה תיראה מבחוץ. איך ניתן שירות נח.
השירות עשוי להיות (וכנראה גם כדאי שיהיה) שונה מאוד מאיך שחשבנו לממש את הפונקציונליות בפועל.
3 מימוש.
כאן אנחנו מממשים את ה API-שהחלטנו עליו. בשלב הזה אנחנו יכולים להוסיף קבועים, או שדות, או שיטות - לפי הצורך. בתנאי --- שהאיברים הללו פרטיים - הם לא חלק מהממשק. לזה קראנו הסתרת מידע, וזהו חלק מהותי בשימור החלוקה שהגדרנו בשלב הראשון.

Enum ב-Java:

Enum (ספירה) הוא סוג נתונים מיוחד המורכב מסט קבוע של קבועים.

הוא משמש לעתים קרובות לייצוג קבוצה של ערכים בעלי שם שקשורים.

- designing good encapsulation with information hiding helps us keep the OCP

```
from enum import Enum

class Day(Enum):
  SUNDAY = 0
  MONDAY = 1
  TUESDAY = 2
  WEDNESDAY = 3
  THURSDAY = 4
  FRIDAY = 5
  SATURDAY = 6

  # Simple function in the enum
  def printDayType(self):
    if self in (Day.SATURDAY, Day.SUNDAY):
      print(f"{self} is a weekend day.")
    else:
      print(f"{self} is a weekday.")

# Print DayType for MONDAY
Day.MONDAY.printDayType()
```

שבוע 3

אינטרפייסים משמשים להגדרת בעצם סוג של חוזים או הסכמים שמחלקות מקבלות על עצמן. בעצם בניגוד למחלקות שמייצגות איזשהו משהו בעולם, או בעולם התוכנה שלנו לפחות, אינטרפייסים לא מגדירים משהו קונקרטי, אלא מגדירים איזושהי תכונה או איזשהו הסכם או איזשהו משהו שמחלקות לוקחות על עצמן.

אינטרפייסים בעצם מדברים על מה עושים וממש לא מדברים על איך עושים את זה. הם לא מדברים על שום דבר שקשור למימוש של המחלקה.

אנחנו אומרים שהממשק הוא מטיפוס גבוה יותר מהמחלקות שמממשות אותו.

ניתן להגדיר משתנים באינטרפייס, באופן מרומז הם public static final.

ההמרה מטיפוס נמוך לטיפוס גבוה נקראת המרה כלפי מעלה אוUpcast.

פולימורפיזם או בעברית רב-צורתיות המצב בו יש מספר סוגים שונים של עצמים שמממשים את הממשק

<u>"תכנות לממשק, ולא למימוש":</u>

העיקרון אומר שתמיד נשאף לכתוב קוד עבור הטיפוס הגבוה ביותר ועם המטרה הכללית ביותר.

אם נקפיד לתכנת לממשק, כלומר לכוון גבוה תרתי משמע אל הטיפוס הכי גבוה ואל מטרה עם מקסימום כלליות, נרוויח המון.

דבר ראשון, נחסוך עבודה של כתיבת קוד דומה.

דבר שני, נשרת גם לקוחות שאנו בכלל לא מכירים או שעדיין לא המציאו.

דבר שלישי, בעתיד נוכל להחליף את המימוש, והלקוח שמתכנת לממשק לא יצטרך לשנות את הקוד שלו. וגם הלקוח שמשתמש בו לא יצטרך לשנות את הקוד שלו, וכן הלאה

עכשיו, לפעמים להכליל את הקוד שלנו זה קל, רק צריך להיות מודעים לזה.

אנקפסולציה מאפשרת תכנות לממשק, אם שברנו תכנות לממשק, כלומר ויתרנו על הכלליות ועשינו משהו אחד עבור מחלקה קונקרטית אחת ומשהו אחר בשביל מחלקה קונקרטית אחרת, כנראה שבדרך שברנו אנקפסולציה .יש לזכור שהאחריות שלנו בכתיבת הממשק הוא לא לקחת אחריות של עצמים אחרים ולא לממש את ההתנהגות שלהם!

תבניות עיצוב: בעיצוב תוכנה, אחד הדברים שנדבר עליהם הוא תרחישים נפוצים שעולים בהרבה תוכנות ופתרונות עיצוב שמתאימים להם. לפתרונות כאלה קוראים תבניות עיצוב.

יתרונות בשימוש בתבניות עיצוב:

- ניתן להשתמש באותה תבנית כללית בפרויקטים שונים.
- תבניות עיצוב נפוצות מכיוון שהן נותנות מענה לבעיות נפוצות.
- מכיוון שמדובר בתבניות נפוצות, הן נבדקו והשתכללו עם הזמן ולכן כדאי להשתמש בהן במקומות המתאימים.

תבניות עיצוב אינן הכרחיות! לא חייב תבנית עיצוב כדי לכתוב קוד בג'אווה

תבניות עיצוב מוכרות: מפעל, סינגלטון, פסאד, מומנטו, דקורטור

מפעל - לוקחים את החלק של היצירה ומייצאים אותו למחלקה ייעודית שהמטרה שלה היא לספוג את האש של בחירת האובייקט הקונקרטי. כל תפקידה ביקום להיות זו שמייצרת את האובייקט. אם היא מייצרת את האובייקט, היא מאפשרת לאחרים לתכנת רק לממשק. בעצם מוציאים את התלות במחלקות המממשות לתלות בממשק.

היתרון בשימוש במפעל: הפרדת הלוגיקה של הייצור והשימוש: הלוגיקה של יצור המופעים נפרדת משימוש בהם.

מפעל הינו דוגמה לאבסטקציה שכן לא אכפת למשתמש כיצד האובייקט נוצר. כמו כן עקרון הסתרת המידע בא לידי ביטוי, הפעם מסתירים את קבוצת הטיפוסים.

Factory- תבנית העיצוב "מפעל" מתכתבת עם עקרון עיצובי ידוע: עקרון הבחירה היחידה (The Single Choice Principle). לפי עקרון הבחירה היחידה, אם יש משהו בתכנית שלנו (זה יכול להיות כל דבר!) עבורו תיתכנה מספר אפשרויות, אז הרשימה המלאה של האפשרויות הללו תופיע רק במקום אחד.

בJava-המרות בין טיפוסים הן חלק חשוב מהשפה. כאשר מתמודדים עם טיפוסים פרימיטיביים (primitive types), ניתן להשתמש בהמרות כדי להתאים את הערך לטיפוס הרצוי.

כמה דרכים של המרות בפרימיטיבים:

המרת טיפוסים פרימיטיביים ב Java-נעשית באמצעות המרה ישירה או יצירת קטע קוד מתאימהם

לדוג, המרת double לint: int n = (int) 5.3;

שיטות ייעודיות:

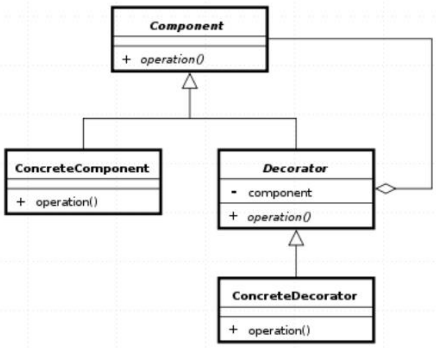
שימוש בשיטות ייחודיות, כמו Math.round() להמרה של double לint-עם עיגול.

תווים שיומרו:

שימוש בסיומת כמו **D** אחרי מספר (**37D**) להצהרה של סוג המספר.

המרה ישירה:

השמת ערך מסוים בטיפוס אחר תוך שימוש בסוגריים (לדוג.(10 / 37 (double) ,



decorator ובעברית "מקשט" זאת תבנית עיצוב (design pattern) שבה מחלקה מממשת ממשק כלשהו, SomeInterface, בעל מתודה :method, ואינטרפייס אחר "יקשט" את המחלקה, כלומר תכיל השיטה של הממשק המקורי(על ידי ירושת אינטרפייסים). המחלקה המממשת תכיל גם אובייקט מהטייפ המקושט(כמעט תמיד יישלח בקונסטרקטור), בפועל המחלקה המממשת את האינטרפייס של הדקורטור תשתמש בשיטות של האובייקט שהיא מכילה, ותוסיף התנהגות 'ייחודית'.

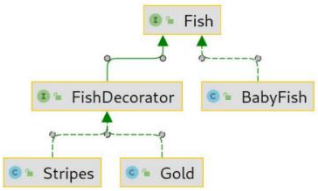
יתרונות הדקורטור:

- עיקרון אחריות יחידה - כל מודול אחראי לפונקציונליות עיקרית אחת.
- יכולת ליצור שילובים חדשים של מחלקות מבלי ליצור מחלקות בפועל.
- אין צורך להתעסק עם ירושה, שעלולה להסתבך.
- יכולת להוסיף פונקציונליות באופן דינמי בזמן ריצה

עקרונות OOP בדקורטור

- *Single Responsiblity* –כל הרחבה בפני עצמה – מתחבר גם לרעיון של פירוק ליחידות קטנות .
- חיזוק ה*אנקפסולציה* (היכולת להסתכל על פיצה עם רוטב קודם כל בתור פיצה או "אתה לא יכול לשחק בהרחבה של סימס בלי המשחק המקורי")

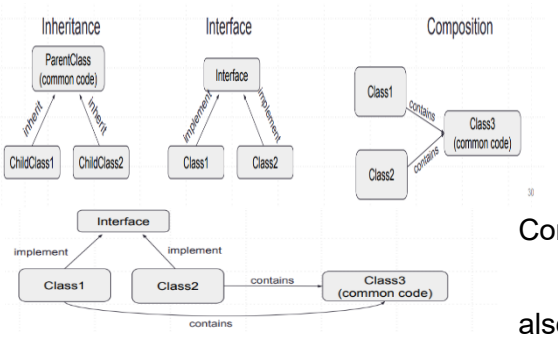
מה שהופך דקורטורים למגניבים באמת הוא שהם יכולים לקשט גם דקורטורים אחרים, ואפילו את עצמם. בעזרת מימוש של מספר קטן מאוד של דקורטורים, אפשר להרכיב קומבינציות שונות ולקבל מספר אקספוננציאלי של התנהגויות חדשות! דקורטורים הם אחד מההיבטים המדליקים ביותר של פולימורפיזם בפרט, ושל תכנות מונחה עצמים בכלל.



שבוע 4

בג'אווה ישנו מנגנון בשם העמסה או Overloading . הרעיון בהעמסה הוא שאפשר לכתוב כמה בנאים אותה מחלקה, אבל עם רשימת פרמטרים שונה. זה תופס גם ל שיטות עם אותו שם.

Inheritance vs Interfaces Inheritance lets us reuse code instead of duplicating it and use Polymorphism.



- Polymorphism & Common behavior → Inheritance.
- Polymorphism without Common behavior → Interface.
- Common behavior without Polymorphism → Composition.
- Polymorphism & Common behavior → Always consider also Interface + Composition.

ב Down-casting-חייבים לעשות את ה-Cast המפורש, ה-Cast ה-explicit, שאומר לנו שאנחנו בטוחים שאנחנו רוצים לקחת משהו שהוא מסוג Animal ולהסתכל עליו בתור Cow.

Down-Casting יכול להצליח.כלומר אם צד ימין, האובייקט הקונקרטי, הוא באמת sub-type של צד שמאל.

עוד דבר שיש להגיד על Down-Casting זה ש-Down-Casting יכול להצליח רק אחרי שעשינו Up-Casting מקודם.

אנחנו לא אוהבים downcasting כי אם ישנו כישלון זה קורה בזמן ריצה! בנוסף לכך ישנה פגיעה בגמישות, כאשר אנו מבצעים downcasting, אנו מטפלים בפחות מקרים. ובפרט לא מתכנתים לממשק.

Overriding	vs. Overloading
Provides the **specific implementation** of a method from its superclass.	Used to increase the **readability** of the program.
Occurs in two classes with an **inheritance relationship**.	Performed **within a class**.
The parameters of the overriding method are the same as the original method.	In case of method overloading, parameters must be different.
Runtime polymorphism.	Compile-time polymorphism.
Return type must be same or covariant in method overriding.	Has nothing to do with return-type.

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
*no modifier**	✓	✓	✗	✗
private	✓	✗	✗	✗

Modifier	Description
Default	declarations are visible only within the package (package private)
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Protected בחבילה אחרת נותן לנו שימוש רק בsuper במחלקה יורשת לא בnew A()

יחידת קומפילציה שאין לה הצהרת חבילה היא חלק מחבילה ללא שם.

- חבילות ללא שם מסופקות על ידי פלטפורמת Java SE בעיקר עבור נוחות בעת פיתוח יישומים קטנים או זמניים או בתחילת פיתוח.

- חבילה ללא שם לא יכולה לכלול חבילות משנה, שכן התחביר של הצהרת החבילה כוללת תמיד הפניה לרמה עליונה בעלת שם החבילה.

שבוע 5

מחלקה אבסטרקטית

- מחלקה אבסטרקטית היא מחלקה שהוכרזה בabstract.
- עשויה לכלול שיטות מופשטות או לא.
- לא ניתן ליצור מופעים ממלקה אבסטרקטית, אך ניתן להרחיב אותם
- ניתן לשמור שדות במחלקה אבסטרקטית

- יש להשתמש במחלקה מופשטת כמו בירושה רגילה, כלומר כאשר יש Is-A קשר בין 2 מחלקות.
- יתר על כן, לא אמורה להיות משמעות ליצירת מופע של מחלקת האב.
- כלומר, זהו טיפוס מופשט (חיה) ולא טיפוס קונקרטי (חתול, כלב).

שיטות דיפולטיביות:

- תאימות לאחור (Backwards compatibility)- לעיתים לא נרצה לממש את כל המתודות בכתיבת מחלקה חדשה.
- שיטות עזר דפולטיות- במצב בו ייתכן שהמשתמש ישתמש בשיטות ספציפיות למימוש הממשק.
- כדי לאפשר ליותר ממשקים להיות "ממשקים פונקציונליים".

שימוש בירושה

- כשיש לנו אובייקט בדרגת בסיס, נניח כלי תחבורה , והיינו רוצים להרחיב את ההתנהגות שלו לסוג קונקרטי יותר, נניח מכונית או מטוס.
- ▪ לכל רכב יש מנגנון תנועה כלשהו, יש לו מושבים וכנראה חלונות.
- ▪ הן במטוסים והן במכוניות, התכונות הללו קיימות, אך חלק מהפונקציות שונות לחלוטין.
- ירושה מוגדרת באופן סטטי, אינה משתנה בזמן ריצה ולכן בטוחה יותר

שימוש בממשק

- כאשר אנו רוצים לציין התנהגות של סוג נתונים מסוים אבל לא מודאגים מי מיישם את התנהגותו.
- מחלקה יכולה ליישם יותר מממשק אחד, לעומת ירושה מאבסטרקטית שמגבילה ל1.

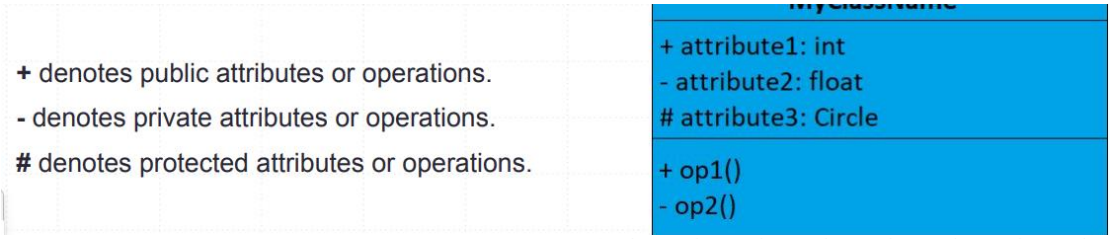
שימוש בהכלה:

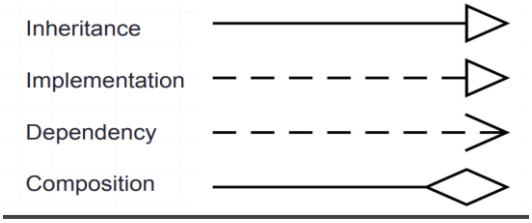
- כאשר ישנו קשר של has-A
- במקום להרחיב מכונית למכונית עם מנוע דיזל, פשוט נכיל מנוע במכונית.
- אפשר להחליף את העצם המוכל בזמן ריצה

חסרונות בשימוש יתר בירושה:

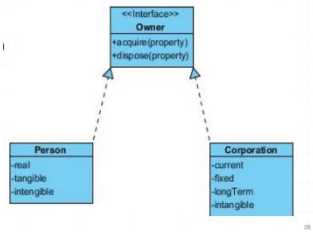
- שרשראות ירושה ארוכות עלולות לייצר API מנופח ומסורבל במחלקות הסופיות.
- לאובייקט ממחלקה סופית יהיו הרבה שיטות הפרוסות על מספר מחלקות שונות.
- קוד שקשה להתמצא בו.

עיקרון Inheritance over Composition : הכלה על פני ירושה. אם ניתן, במקום לרשת ממחלקה, נכיל מופע שלה באופן ששומר על הAPI.

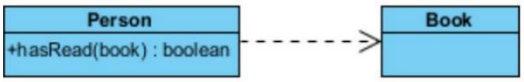




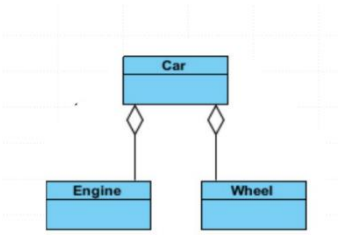
:Inheritance
- כל מופע של המסווגן הספציפי הוא גם מופע עקיף של המסווג הכללי.
- מייצג מערכת יחסים "is a ".
- שם מחלקה מופשטת מוצג באותיות נטוי.



:Implementation
- יישום הוא קשר בין הממשק למחלקה המכילה את פרטי רמת היישום שלה.
- לממשק יש את הסימון <<ממשק>> לפני השם.



:Dependency
- אובייקט של מחלקה אחת עשוי להשתמש באובייקט של מחלקה אחרת בקוד של מתודה. אם האובייקט אינו מאוחסן באף שדה, אז זה מדגם בעל יחסי תלות.



:Composition
- מייצגת מערכת יחסים "has-a".

דוגמא:
- אובייקט מסוג Decorator מאציל רכיב אובייקט כחבר נתונים.

<u>אסטרטגיה:</u> החלפת התנהגות בזמן ריצה, בהנתן אוסף אלגוריתמים\התנהגויות בעלות API זהה נבחר התנהגו\יות שיבחרו וישומשו.

נבנה מערכת אסטרטגיות באופן הבא:

1. יצירת ממשק\מחלקה אבסטרקטית.
2. יצירת מחלקו קונקרטיות שתטמיע את הAPI הנדרש.
3. מחלקה שתשתמש בעצם של האינטפרייס\המחלקה האבסטרקטית בלי להכיר את המימוש.

יתרונות:

- הסתרת מידע - הלקוח שמשתמש לא חייב לדעת איזה אלגוריתם פועל בתוך "הקופסה השחורה".
- מחזור קוד - הלקוח יכול להשתמש באסטרטגיה מסויימת גם במקומות אחרים בקוד.

עקרונות OOP באסטרטגיה

- *Open-Closed* ניתן להוסיף אסטרטגיות נוספות מבלי לשנות את הקיימות
- *Single-Choice* –הבחירה באסטרטגיה מתבצעת במקום אחד .
- *פירוק לחלקים קטנים* – כל אסטרטגיה מתנהלת באופן עצמאי.

נשתמש באסטרטגיה ולא בירושה כאשר:

- תנאי is-A אינו רלוונטי, למשל מתודת מיון בcollections, הרי שsort אינו collections
- הסתרת מידע- הלקוח שמשתמש לא חייב לדעת איזה אלגוריתם פועל בתוך "הקופסה השחורה".
- מודולריות- פיצול מימוש של אלגוריתמים למקום אחר בקוד.
- מחזור קוד - הלקוח יכול להשתמש באסטרטגיה מסויימת גם במקומות אחרים בקוד.
- שינוי בזמן ריצה, ניתן לשנות אסטרטגיה בזמן ריצה לעומת ירושה שהינה סטטית.

Javadoc

JavaDoc הוא כלי ליצירת תיעוד שאינו חובה עבור פרויקטי Java. ניתן להסביר ולתיעוד קוד בצורה מובנית וקריאה. לשם כך, משתמשים בתגי JavaDoc כמו @return, param ו-@version כדי לתאר פרטים של שיטות וממשקים. ב-IntelliJ IDEA, ניתן ליצור תיעוד באופן אוטומטי ולייצא אותו ל-HTML באמצעות כלי Generate JavaDoc.

שבוע 6

מחלקה סטטית מוגדרת באמצעות final class A, לא ניתן לרשת ממנה.

משתנה סטטי הינו שדה שלא מייצג תכונה של המופע, אלא של המחלקה. יש רק עותק אחד עבור כל המופעים של המחלקה.

שיטה סטטית אינה יכולה לגשת למשתנים שאינם סטטים. אין הגיון בשימוש בthis במתודה סטטית.

כל המחלקות בmath הינן סטטיות

משתנה final הינו משתנה שאינו משנה את ההצבעה שלו לאורך התכנית, כלומר משתנה שאינו "נע".

שיטה final אינה ניתנת לדריסה.

כשיש יותר משתנים נעים התכנית קשה יותר לתכנון, מצריכה יותר תחזוקה, מתקלקלת יותר ופחות יעילה.

שדה סטטי מיוטבילי נע בחופשיות דרך כל המחיצות בקוד ומהווה בדרך כלל קוד גרוע, שקשה לעקוב אחריו.

שימוש נכון בשיטות סטטיות מצמצמות את המיוטיבליות של השדות מחלקה.

איזה סוגי איברים יכולים ממשקים להכיל? בראש ובראשונה, שיטות ללא מימוש (שבמחלקה הממשת 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 שדות סטטיים סופיים.

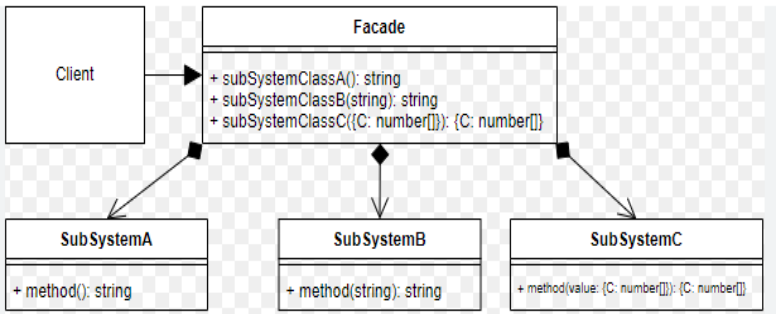
Shadowing- דריסה באמצעות ירושה של משתנים\שיטות

נשים לב כי בירושה data members (גם הסטטיות) והמתודות הסטטיות נקבעים לפי ה reference type ותוכן השיטות(גם סטטיות) לפי הobject הקונקטי.

Facade- תבנית הFacade מדברת על לקחת מערכת מורכבת ולהסתכל עליה דרך חלון צר או דרך איזשהו אינטרפייס קטן ויותר פשוט.

Facade הוא בעצם רלוונטי כשיש לנו איזושהי מערכת גדולה, עם הרבה מאוד מחלקות ויש שם הרבה תלוית בין מחלקות שונות. שזה בעצם יוצר לנו איזשהו API שהוא מורכב וקשה לעבוד איתו. אבל בהרבה מאוד מקרים, יש לקוחות שלא צריכים לדעת את כל המורכבות של המחלקה, אלא מספיק להם לדעת חלק מאוד קטן ממנה או איזשהו API פשוט יותר. בעצם זה שהם חשופים לכל הAPI המורכב הופך את המערכת הזאת לפחות אטרקטיבית בעיניהם, כי יותר קשה לעבודה איתה, לא בטוח שמשתמשים בה כמו שצריך, יש שם כל מיני בעיות.

ה API-של מחלקת ה façade-אינו בהכרח תחליף ל API-המקורי של המחלקות המרכיבות אותו ולכן ישנן מקרים בהם ניתן לעשות שימוש ב API-של המחלקות המקוריות, ככל שיש בכך צורך.



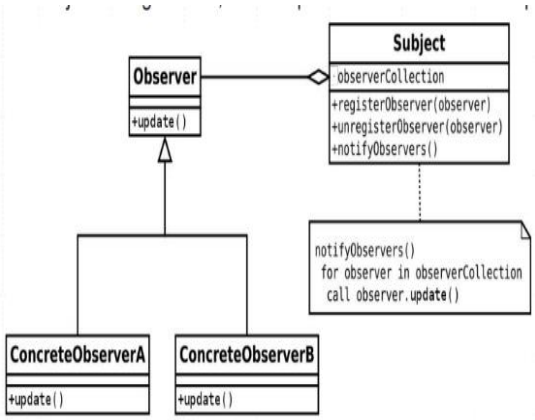
יתרונות הfacade:

- API פשוט ונוח למשתמש
- לקוח לא מודע לשינויים שמתרחשים מאחורי הקלעים.
- מחלקת הfacade אינה פוגעת במחלקות שהיא "מפשטת" אלא רק משתמש בהן.

עקרונות OOP בfacade:

- API מינימלי – חושפים רק חלק מהמידע ללקוח.
- הסתרת מידע – אפשר לחשוף רק חלקים מסויימים מה API-של המחלקה וכך להסתיר חלקים מהמימוש.

Observer



תבנית האובסרבר מגדיר תלות של אחד לרבים בין אובייקטים כך מתי אובייקט אחד משנה מצב, כל התלויים בו מקבלים הודעה ומתעדכנים אוטומטית.

מבנה:

1. ליצור ממשק subject שיכיל את השיטות הוספה והודעה לאובסרברים
2. ליצור מחלקה שתממש את הממשק subject
3. ליצור ממשק observer שתהיה לו שיטת עדכון
4. ליצור מחלקות שיממשו את ממשק הobserver

Singleton

תבנית הסינגלטון היא תבנית עיצוב תוכנה המגבילה את המופע של מחלקה למופע בודד.

- שימושי כאשר יש צורך בדיוק באובייקט אחד כדי לתאם פעולות על פני המערכת.
- נשים לב שבתבנית הסינגלטון, הבנאי הוא פרטי, והוא מחזיק בהתייחסות לאובייקט יחיד.
- נשתמש בסינגלטונים כאשר נרצה משתנה גלובלי משותף יחיד.

לדוגמה, אובייקט בודד שיכול להתחבר למסד נתונים, כל מחלקה יכולה לגשת לחיבור, אך אין צורך ליצור מופע עבור כל מחלקה.

לחלופין, נשתמש בסינגלטון כאשר האפשרות של יותר ממקרה אחד עלולה לשבש את ההיגיון של התוכנית.

השלבים:

1. בנאי פרטי להגביל את יצירת המחלקה ממחלקות אחרות.
2. משתנה סטטי פרטי מאותה מחלקה שהוא המופע היחיד של המחלקה.
3. שיטה סטטית ציבורית שפועלת בתור בנאי, ומחזירה את המופע של המחלקה.

סינגלטון מהווה בדיוק שדה מיוטבילי משותף שכל כך פחדנו ממנו. לכן יש להזהר בשימוש בו.

לסיכום:

יתרונות	חסרונות
וודאות כי נוצר אובייקט יחיד	מסובך קשה לתחזוקה והקוד לעיתים לא מובן
גישה גלובלית לאובייקט הספציפי	מה קורה אם הרבה מנסים לתקשר עם האובייקט יחד?
חסכון בזמן ומקום.	קשה לעשות טסטים לclient של הסינגלטון

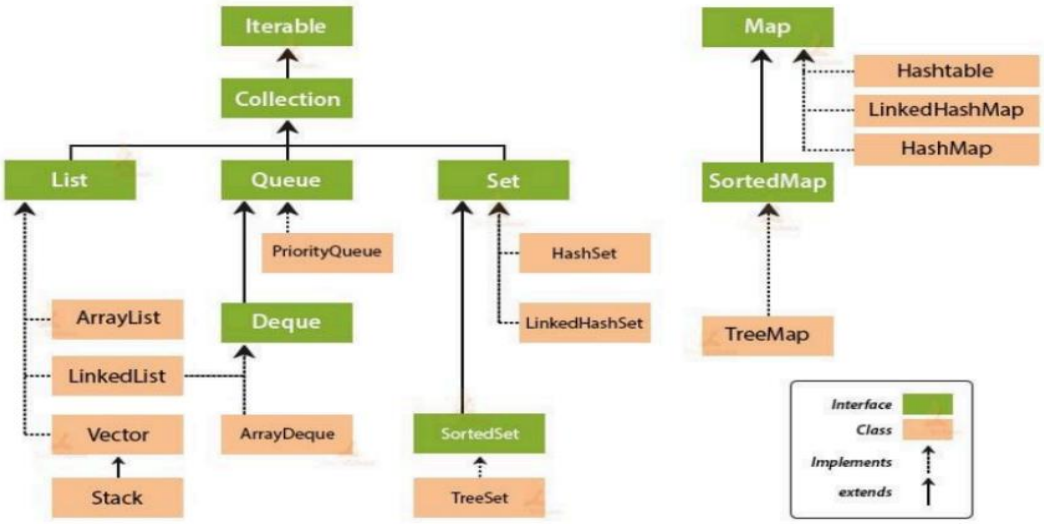
סינגלטון מול מחלקה סטטית

שניהם מרגישים דומים מאוד - ומקפידים על עותק אחד בלבד.

- אבל, סינגלטון יכול לעשות שימוש בירושה וממשקים.
- נזכור קוד לממשק לא יישום!

שבוע 7

חבילת הCollections שלjava, מכילה ממשקים רבים, היא בעצם חבילת מבני הנתונים שלjava . Collections הוא data stracture כללי, יש לו כל מיני מתודות שהוא מגדיר כמו למשל add remove size. יש גם ממשקים יותר ספציפיים ב Collection-כמו למשל List שיורש מהממשק Collection, מייצג מבנה נתונים שיש בו מיקום על אינדקס. הסיבה ש Collection-וגם List הם ממשקים ולא מחלקות, היא שהממשקים של Collection הרבה פעמים מדברים יותר על המה ולא על האיך. בדוגמה של הממשק :List יש הרבה מאוד דרכים לממש רשימה - רשימה מקושרת, רשימה מבוססת מערך וכו .'לכל האינטרפייסים האלה יש את אותו API , המתכננים של java החליטו שיותר הגיוני להגדיר את ה Collections-האלה בתור אינטרפייסים - כלומר ממשקים ולא מחלקות שאפשר לרשת מהן.



ממשקים בcollections:

- List-סדרת איברים
- Set ממדל את הקונספט של קבוצה מתמטית. רצף של איברים ללא כפילויות וללא סדר.
- Map- מייצג מבנה נתונים שממפה בין מפתחות לערכים. המפתחות חייבים להיות ייחודיים.

מימושים בcollections:

- *ArrayList* מימוש List באמצעות מערך. *LinkedList- מימוש List בעזרת רשימה מקושרת*
- *HashSet מימוש של set בעזרת טבלת גיבוב(עם מנגנון התנגשויות), TreeSet*
- *HashMap* מימוש של Map באמצעות טבלת גיבוב של המפתח, TreeMap

Genericב java-מאפשר לנו להגדיר פרמטר אחד או יותר לכל מחלקה או לכל אינטרפייס.

public int hashCode()

public boolean equals(Object o)

סטרינגים עם אותו ערך יחלקו אותו ערך האש`

המימוש של equal ב set למשל מסתמך על ה hashcode שמומש.

Contains על טבלת גיבוב ישתמש בhashcode כדי לגשת לאינדקס בטבלה, שם תעבור על כולם ותשתמש בequals

אובייקטים שווים חייבים להחזיר את אותו hashcode!

	Insert	Find	Remove	Access by index	implements
ArrayList	O(1)*	O(n)	O(n)	O(1)	List
LinkedList	O(1)	O(n)	O(n)	O(n)	List
HashSet	O(1)*	O(1)*	O(1)*	-	Set
TreeSet	log(n)	log(n)	log(n)	-	NavigableSet
HashMap	O(1)*	O(1)*	O(1)*	-	Map
TreeMap	log(n)	log(n)	log(n)	-	NavigableMap

ברוב הפעמים, השגיאות בתכנות נחלקות לארבעה סוגים עיקריים:

1 שגיאות קומפילציה (Compilation Errors): אלו שגיאות שמתרחשות בזמן קומפילציה של הקוד, והן מופיעות במהלך כתיבתו.

דוגמה: חסר נפיל (semicolon), שגיאה בתחביר, פעולה עם משתנה לא מוגדר, וכדומה.

2 שגיאות זמן ריצה (Runtime Errors): שגיאות שמתרחשות בזמן ריצה של התוכנה, והן נגרמות ממצבים לא צפויים בזמן הביצוע.

דוגמה: התרחשות NullPointerException, גישה לאינדקס לא חוקי במערך, וכדומה.

3 באגים ידועים (Logical Errors): לא בעיה בתחביר, המתכנת יודע עליהן. התוכנה רצה כרגיל, אך פעולתה או תוצאתה אינן כפי שרצוי.

דוגמה: שגיאה בלוגיקת הקוד, חישובים שגויים, ותוצאות לא צפויות.

4 שגיאות בזמן ריצה של המערכת (System-level Runtime Errors): באגים שהם לא בעיה בתחביר ,ושהתוכנה לא זיהתה ושגם אנחנו לא זיהינו ,אבל הם עדיין קיימים בתוכנה שלנו .יכול להיות שפשוט לא שמנו לב ,ויכול להיות שעוד לא קרה התרחיש שהיה מציף אותן בתור שגיאות מסדר שני או שלישי.

דוגמה: חסרון הרשאות לקובץ או לתיקייה, תקשורת רשת נכשלת, וכדומה

המרת שגיאה מסדר 4 לשגיאה אחרת קורית באמצעות טסטים.

על מנת להמיר שגיאה מסדר שלישי ורביעי לשגיאה מסדר שני נשתמש בassert

על מנת להמיר שגיאה לשגיאה מסוג 1 נשתמש באזהרות קומפילציה, שימוש בשפה סטטית(שפה בה הישויות המחלקות, היחסים בין הישויות... מוגדרים היטב בזמן כתיבת הקוד, למשל JAVA) או השלמת קוד חכמה.

מנגנון ה Generics-הוא דוגמה מעולה למנגנון שמטרתו להמיר חלק משגיאות זמן הריצה לכאלו שיעלו כבר בזמן הידור.

חריגות exceptions:

יתרונות: קריאות של הקודת חסכון בכתיבת קוד, שיטת עבודה מסודרת והגדרת קונבנציה, הפרדה בין קוד שגיאות לקוד התוכנה עצמה.

ישנן 3 סוגי שגיאות עיקריות-

Error- קשור יותר לרמת המערכת פחות לקורס שלנו אינו מטופל על ידי המתכנת\משתמש.

Checked exceptions: יורשות מ Expection. מתרחשות כתגובה לשגיאות משתמש וקלט לא תקין. מצריכות throw וגם טיפול בצורת try\catch\ throw,

Unchecked exceptions- יורשות מ runtimeExpection ובדרך כלל נגרמות כתוצאה מטעות של המתכנת. אין צורך בזריקה.

אם נכנסו לcatch לא נכנס לבאים אחריו לכן יש חשיבות להיררכיה בכתיבה

<u>הערה</u> סוג השאלות כמו בחלק האחרון של 7.3 יכולות מאוד להיות במבחן,

class VerySpecificIOException extends IOException{

try{ throw new VerySpecificIOException(); } catch (IOException Ve){ } }

<u>מחלקות מקוננות</u>

מחלקה שהוכרזה בתוך מחלקה אחרת. member של המחלקה החיצונית. ניתן להכריז בכל סוגי הנראות.

למה להשתמש במחלקות מקוננות?

- זוהי דרך לקבץ באופן הגיוני מחלקות המשמשות רק במקום אחד:

  במקרים בהם מחלקה שימושית רק למחלקה אחת אחרת.

- מגביר את האנקפסולציה:

  נניח וישנן שתי מחלקות ברמה העליונה, A ו-B. ונניח ש-B זקוק לגישה לmembers ב-A שאחרת היו מוכרזים כפרטיים.

  על ידי הסתרת מחלקה ב' בתוך מחלקה א', ניתן להכריז על ה membersשל א' כפרטיים, ו-B יכול לגשת אליהם.

  בנוסף, ניתן להסתיר את ב' עצמו מהעולם החיצון.

- תורם לקריאות

  קינון של מלקות קטנות בתוך מחקות ברמה העליונה ממקמת את הקוד קרוב יותר למקום שבו הוא משמש

ישנן שתי קטגוריות של מחלקות מקוננות

<u>מחלקות מקוננות סטטיות Static Nested Classes:</u>

- מופע של מחלקה סטטית-מקוננת שייך למחלקה ולא למופע מחלקה חיצונית ספציפית.
- אין לו גישה לmembers שאינם סטטיים של המחלקה החיצונית.
- בהינתן מופע של המחלקה החיצונית, המחלקה הסטטית יכולה לגשת חבריה הפרטיים.
- אין צורך ביצירת מופע של המחלקה החיצונית על מנת ליצור אובייקט.
- מחלקה סטטית יכולה ליצור מתודות שאינן סטטיות.

<u>מחלקות פנימיות (מחלקות מקוננות לא סטטיות) Inner Classes:</u>

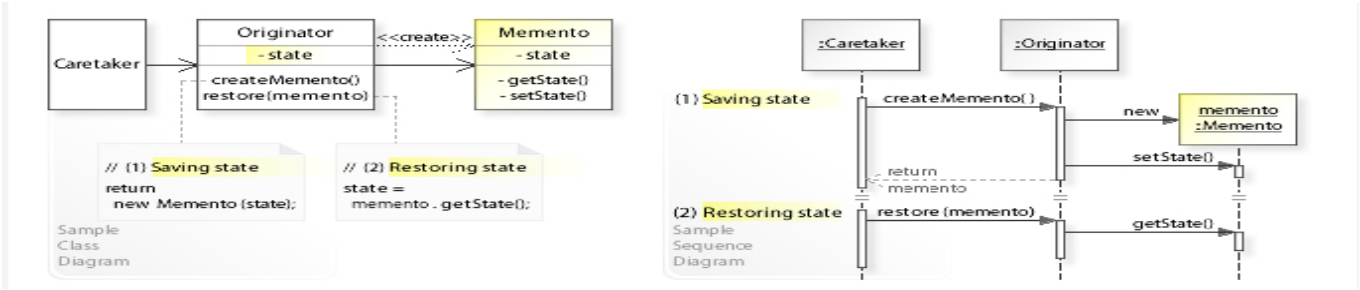
- יש ליצור מופע של המחלקה החיצונית כדי ליצור מופעי עבור המחלקה הפנימית.

- יכולה לגשת לכל ה members במופע החיצוני שלו.
- אינה יכולה להגדיר מתודות סטטיות

<u>מתי להעדיף איזה סוג של קינון?</u>

- כאשר אין צורך לשייך את מופעי המחלקות המקוננות למופע מקיף ספציפי נשתמש במחלקה מקוננת סטטית.
- כאשר נרצה לשייך את המחלקה המקוננת למופע מקיף, נשתמש במחלקה Inner.

<u>תבנית momento</u>



Momento- מחלקה שאחראית על יצירת המומנטו ושחזור (מחלקה פנימית ל originator) (snapshot)

Originator- מחלקה שיודעת להחזיר עותק של מצב או לשחזר מתוך מצב קיים.(code project)

CareTaker- המחלקה שמחזיקה את המצב הנוכחי, מכילה אובייקט של Originator על מנת שתוכל לעדכן או להחזיר את המצב הנוכחי למצב אחר. (git)

שבוע 8

דוגמות לממשק פונקציונלי

Predicate<T>: boolean test(T t);

Supplier<T>: T get;()

Consumer<T>: void accept(T t);

Function<T, R>: R apply(T t);

BiPredicate<T, R>: boolean test(T t, R r);

דוגמה ללאמבדה:(a,b)->{a+b;};

דוגמה לmethod reference Animal::speak

כאשר בפונקציית למבדה משתמשים במשתנים / רפרנסים חיצוניים, המשתנים החיצוניים חייבים לקבל השמה רק פעם אחת.

מחקלה אנונימית הינה מחלקה מקוננת שאין לה שם.

מוצהרת ומוגדרת בתוך הביטוי המייצר אותה.

- מיישמת ממשק או מרחיבה מחלקה קיימת

```
interface SomeInterface {
   public void method();
}

class OuterClass {

   // defining anonymous class
   SomeInterface anonymousClass = new SomeInterface(){
       // body of the anonymous class
       @Override
       public void method() {
           // implementation
       }
   };
}
```

	Initialization	Access	Usage
static nested class	`OuterClass.NestedClass obj = new OuterClass.NestedClass();`	All static members of the outer class	When the nested class doesn't need access to a specific instance of the outer class throughout its lifetime
Inner class	`OuterClass outerObj = new OuterClass();` `OuterClass.InnerClass innerObj = outer.new InnerClass();`	All members of the outer instance.	When the nested class needs access to a specific instance of the outer class throughout its lifetime
Local class	Defined within a method and instantiated within the same method.	All members of the outer instance and local variables of the enclosing method.	When the nested class only needs to be used within the scope of a single method, and it is not necessary to expose it to the rest of the program.
Anonymous class	Defined and instantiated in a single expression.	All members of the outer instance and local variables of the enclosing method.	To create a short class that is only needed in a single context.

	Anonymous Class	Lambda Expression
Definition	An inner class without a name.	For declaring independent methods without a name.
From the compilers view	The compiler creates a separate '.class' file for every anonymous class.	Lambda expressions aren't converted to **.class** files. ⇒ saves time and memory.
when to use	When there is more than one abstract methods to implement.	For implementing functional interfaces.
Implementation	We need to write the class body.	We only need to provide the **function body**.

Callback- פונקציה שנשלחת כפרמטר וניתנת לשימוש, מממשת ממשק פונקציונלי. כלומר זהו פרמטר שהטיפוס שלו היא פונקציה

במקרה של ירושה, מחלקה אחת יורשת ממחלקה אחרת וישנם התנהגויות קבועים שמוגדרים בצורה קבועה. במקרה של אסטרטגיה, ההתנהגויות הם דינמיות וניתנים להחלפה בזמן ריצה.

השימוש בירושה מצריך הגדרת קבועה מראש של התנהגויות, בעוד שהשימוש באסטרטגיה מאפשר גמישות רבה יותר בהחלפת התנהגויות בזמן ריצה.

בנוסף, הטכניקה של קולאבקס (Callbacks) מתייחסת להפעלת פונקציות או קטעי קוד בתגובה לאירועים מסוימים בתוך התוכנית, מה שמאפשר גמישות רבה והתמקדות בתגובה לאירועים מסוימים.

לסיכום, השימוש בין ירושה, אסטרטגיה וקולאבקס הוא תלוי במהות הבעיה ובדרישות הספציפיות של הפרויקט.

יתרונות הcallbacks

- נוחות - קיבלנו פתרון עם כל היתרונות של אסטרטגיה, ולא רק שהוא פחות מסורבל מאסטרטגיה, הוא פחות מסורבל אפילו מירושה.
- ניתן לשנות את האסטרטגיה בזמן ריצה.
- אין הגבלה על ידי עץ ירושה נוקשה.
- ניתן להגדיר כמה Callbacks לאותו GameObject, כל אחת מטפלת באיזושהי סוגיה אחרת.
- קוד קצר וברור יותר.
- מודולריות שבה אני שולט בנפרד באסטרטגיות ובנפרד ב-GameObject.
- ניתן גם לשלוח את אותן Callbacks לעצמים מסוגים אחרים.
- API קל יותר להבנה

Functional Programming Vs. OOP

Functional Programming:

- כאשר הפלט תלוי בקלט ולא במצב של אובייקט.
- הקוד הינו יעיל נקי ומודולרי

לסיכום:

- כאשר נרצה לבצע פעולות כל ישויות ולהרחיב יישום על ידי הוספת ישויות נשתמש בOOP
- כאשר התוכנית מורכבת בעיקר מפעולות וישויות בודדות בלבד נשתמש בממשק פונקציונלי

תרגול 9

תכנות גנרי

גנריות מאפשרות לנו ליצור מחלקה, ממשק ושיטה בודדים שניתן להשתמש בהם עם סוגים שונים של נתונים (אובייקטים).

- גנריקה מבטיחה בטיחות type. שגיאות המתרחשות מגנריות קורות בזמן ההידור.
- שגיאת המרה(ללא generic) היא שגיאת זמן ריצה; באמצעות שימוש גנרי אנו מקבלים את הכוח להביא את זה לשגיאת קומפילציה.

Raw type הינו שם של ממשק או מחלקה גנרית ללא פרמטר כלומר לא נשתמש ב<>.

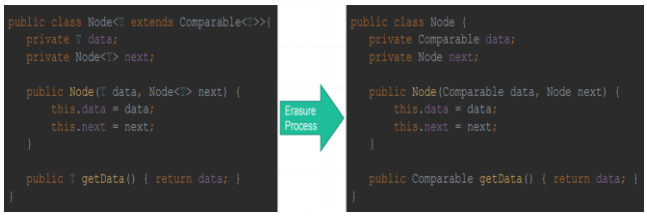
הקוד יתקמפל אך מהווה bad practice ויש להמנע מכך

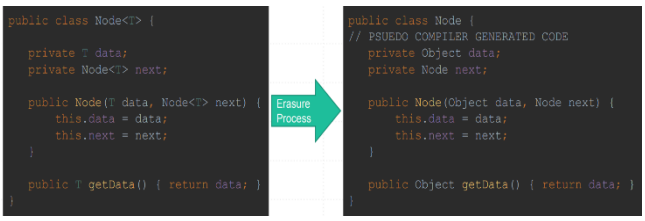
בעצם זה מבטל את כל השימוש בפרמטר מסוג type

הפתרון? שימוש בסוגריים <Int> לאחר הצהרת המשתנה (קצת מפגר אם שכחתי לשים פרמטר למה שאני לא אשכח סוגריים עם פרמטר)

נשים לב! לא ניתן לייצר מערך [] עם אובייקטים גנריים, אך כן ניתן להשתמש ב collectionsלשם כך.

Erasure בתהליך הקומפילציה נמחקים כל הטייפים הגנריים, ומוחלפים ב Object-או בחסם העליון.





ירושה גנרית: class SubClass<T> extends SuperClass<T>

Upper bound: Class Zoo<T extends Animal>

כשנרצה שהטייפ הגנרי T יהיה מטייפ מסויים או יורש\מממש אותו

ידוע כי הממשת הינה T לכן ניתן לקרוא לT ומעלה (בין היתר Animal) ולכתוב T ומטה.

Wildcard כאשר בתוך הטייפ יש <?> נוכל רק להוסיף null או לקרוא ממנו לobject

Wildcard upper bound: List<? extends Animal> list = new LinkedList<Dog>();

נשתמש ב wildcard upper bound ולא ב upperbound רגיל כשאין שימוש בטייפ הספציפי.

ידוע כי הממשת Animal ומטה לכן ניתן לקרוא לAnimal אך להוסיף null בלבד.

Wildcard lower bound List<? super Animal>

ידוע כי הממשת הינה Animal ומעלה לכן ניתן לכתוב Animal אך לקרוא רק לObject שכן כל המחלקות יורשות ממנה.

פעולה:	*<T extends Animal>*	*<? extends Animal>*	*<? super Animal>*
קריאה:	ניתן לקרוא מהמערך לתוך **T ומעלה**	ניתן לקרוא מהמערך לתוך Animal **ומעלה**	ניתן לקרוא רק לתוך **object**
הוספה:	ניתן להוסיף **T ומטה** או **null**	ניתן להוסיף **null** בלבד	ניתן להוסיף **Animal ומטה** או **null**

עקרונות OOP

אנקפסולציה- לאגד ליחידה אחת את כל הפונקציות וגם את כל המשתנים שעוסקים באותו תחום אחריות.

- הסתרת מידע- כל עצם מפריד בין מה שמשרת יחידות אחרות ומה שנועד לשימוש פנימי
- תכנות לממשק ולא למימוש - העיקרון אומר שתמיד נשאף לכתוב קוד עבור הטיפוס הגבוה ביותר ועם המטרה הכללית ביותר.

אבסטרקציה היא התרחקות מהמימוש והתקרבות לצורך של הלקוח. הרעיון הינו להסתיר מידע מיותר בקוד שלנו ולהראות רק את מה שחיוני.

- API מינימלי- האובייקט צריך לעשות כל מה שמצופה ממנו, אבל הוא גם צריך לעשות **רק** מה שמצופה ממנו, וכל המוסיף גורע.
- אבסטרקציה+אנקפסולציה -> *Open-Closed* – הקוד פתוח להרחבה ללא צורך בשינוי

פולימורפיזם או בעברית רב-צורתיות המצב בו יש מספר סוגים שונים של עצמים שמממשים את הממשק

- Single responsibility- לכל מחלקה יש תפקיד אחד בלבד.

- *Single-Choice* –אם יש משהו בתכנית שלנו עבורו תיתכנה מספר אפשרויות, אז הרשימה המלאה של האפשרויות הללו תופיע רק במקום אחד.