מבוא לתכנות מונחה עצמים – Oop

תוכן עניינים

1. הקדמה

מטרות הקורס:

1. עקרונות בתכנות מונחה עצמים.
2. המושגים: desgin , desing patterns
3. מימוש פרקטי של הקורס מבני נתונים.
4. תכנות ב JAVA.

מבנה השיעורים הפרונטליים: שיעורים מקוונים מחולקים ל 4-8 חלקים כאשר כל חלק הוא בין 5 ל 15 דק'.

סילבוס:

הרצאות 1-2: introduction to java

הרצאות 3-5: polymorphism and basic design

הרצאות 6-7: core topics in java

הרצאות 8-9: modularity and advanced design

הרצאות 10-13 : advanced topics.

תכונות של תוכנה טובה:

מצד המשתמש: עונה על הדרישות, עובדת, מהירה ויעילה, אינה קורסת הן מפני טעויות שלה והן מפני טעויות של המשתמש, compatible.

מצד המתכנת: קלה לקריאה, לדיבוג ולהבנה. קלה לשימוש בצרכים אחרים וקלה לשדרוג.

תכנות מונחה עצמים – שיטת תכנות המשמשת לבניית תוכנות גדולות בעל מבנה העונה על הדרישות שהזכרנו שורה קודם.

מושגים בסיסיים:

1. Objects and Classes
2. Encapsulation
3. Inheritance
4. Polymorphism
5. Genericity

תכנות מונחה עצמים ותכנות פרוצדוראלי

התכנות הפרוצדוראלי מורכב מרצף פעולות ואילו האו"פ הוא פעולות בין אובייקטים.

אובייקטים

אובייקט בתוכנה דומה לאובייקט ב"עולם האמתי", לאובייקט ישנו מצב והתנהגות.

אובייקט בתוכנה יודע להחזיק מידע (מצב פנימי – data members) ופעולות (methods) כאשר הפעולות הן "האינטראקציה של האובייקט עם העולם".

ההבדל בין תכנות מונחה עצמים לפרוצדוראלי – באו"פ אנחנו פונים לפעולות דרך האובייקט עצמו לעומת הפרוצדוראלי שבו אנחנו שולחים את האובייקט אל פונקציה והיא מבצעת עליו את הפעולה.
בתכנות מונחה העצמים זה ישנם שני יתרונות:

1. קל להבנה.
2. קל לשדרוג שהרי מבנה המתודות כבר קיים ולכן הוספת אובייקט אינה משנה מהותית את התוכנה שהרי המבנה הבסיסי כבר קיים.

הרצאה 1

מחלקות - Classes

המשותף לאובייקטים הוא שיש להן את **אותו סט** של תכונות (Data Members) ופעולות (Methods) אך **הערכים** של התכונות הוא **שונה**.

מושגים:

Class - מייצגת משפחת אובייקטים בעלי תכונות ופעולות זהות.

Instance - מופע של מחלקה, אובייקט במחלקה.

Methods - פונקציות המקושרות למחלקה מסוימת.

- מתודה יכולה לגשת לdata member הספציפי של כל אובייקט.

Constructor - מתודה היוצרת את האובייקט ומאתחלת אותו.

- תכונות קונסטרקטור:
  - שמו זהה לשם המחלקה, יכול קבל פרמטרים לאתחול, אין לו ערך החזרה.
- על מנת ליצור אובייקט חדש יש להשתמש במילה new ואז שם הקונסטרקטור.

ענייני זיכרון: קונספט מחלקה מוגדר פעם אחת אך כל אובייקט תופס זיכרון משל עצמו.

types: primitives and reference

משתנים בג'אווה:

- Reference
- primitive

primitives - מחזיקים את סוגי המידע הנפוצים והבסיסיים ביותר שיש: int, double, char, boolean.

כל ה primitives מאותו הסוג דורשים את אותה כמות זיכרון.

reference - אינו אובייקט ממש אלא מצביע לאובייקט.

לדוג' Bicycle bike1 = new Bicycle(1) השמאלי הוא רפרנס והימני הוא האובייקט עצמו. סימן השווה יוצר רפרנס למצביע חדש ולא יוצר אובייקט חדש, ולכן שינוי אחד מהם ישנה את השאר. ז"א Bicycle bike2 = bike1. הרפרנס החדש bike2 אינו אובייקט אלא מצביע לאובייקט ש-bike1 מצביע עליו ולכן שינוי שייעשה ב bike2 ישפיע על האובייקט המצביע על bike1.

Garbage_collector – מוחק אובייקטים (והנתונים שלהם) לאחר שאובייקט חדל מלהיות שימושי (לא שימושי משמעותו שאין אף pointer המצביע עליו).

Class_String – במחלקת ה-String ניתן להשתמש בסימן " = " בשביל להגדיר אותה. ואין צורך לקרוא לקונסטרקטור. ולכן String myStr = "hello" הוא בעצם יצירת אובייקט חדש ממחלקת סטרינג. ולמחלקה זו יש מתודות מסוימות. כמו האופרטור [] או getCharAt() וכו'.

String הינו immutable – פרטים באובייקט עצמו אינם ניתנים לשינוי (אך כמובן שהרפרנס עצמו יכול להשתנות – כלומר להצביע על אובייקט String אחר).

קבועים - חלק מהדיזיין הוא יצירת הקבועים – קונסטים. מטרת הקבועים היא למנוע טעויות משתמש בשינוי ערכים שלא נועדו להשתנות בתאם למטרת הקוד הראשונית.

בג'אוה זה נעשה באמצעות המילה השמורה final הנכתבת לפני יצירת האובייקט.

הערה: ניתן ליצור השמה באובייקט קבוע רק בעת האתחול.

ההבדל בין final ל immutable - final מתייחס למצביע של האובייקט ולכן המצביע אינו יכול להשתנות, כלומר אינו יכול להצביע על אובייקט אחר. אך immutable מתייחס לאובייקט עצמו שאינו יכול להשתנות, כלומר פרטיו אינם יכולים להשתנות.

הרצאה 2

נושאי ההרצאה

1. Scope
2. Instance vs static
3. Minimal API
4. Information Hiding

Scope

Local Variables - משתנה המוגדר בתוך מתודה (בכל מקום בתוך המתודה).
משתנה לוקאלי יכול להיות מוגדר בתור primitive או reference וכן יכול להיות קבוע.
Scope - קטע קוד המונח בין שני סוגריים מסולסלים.
Scope מגדיר נראות ובלתי נראות של משתנים. משתנים לוקאלים נגישים מסקופים שלהם או פנימיים להם אך לא בסקופים מחוץ להם.
Name Space Pollution – עיקרון תכנות שבו מגדירים משתנים ככל הניתן בסקופ השייך אליהם - בסקופ שבו מתעסקים עם המשתנה ולכן רצוי להגדיר משתנה פנימי סמוך ככל הניתן לשימוש בו. אמנם במקרים מסוימים נעדיף להגדיר משתנה בסקופ חיצוני כדוגמת סקופ של לולאת פור ובכך נמנע את הגדרת המשתנה מחדש בכל איטרציה בלולאה.

Static

Static Members - משתנים הקשורים למחלקה ולא לאובייקט באופן ספציפי. נקראים גם Class Variables לעומת Instance Variables.
הדרך הנכונה להשתמש במשתנים סטטיים היא באמצעות קריאת למחלקה: ClassName.StaticVariable (בלומר אפשר גם דרך ה-instance אך רצוי דרך המחלקה עצמה).
Static Methods – מתודות הפועלות על כלל המחלקה, ולכן מתודה סטטית יכולה לשאול רק מה הם הערכים של static members אחרים ולא של instance members. מטרתם לקחת אלגוריתם שאינו קשור לאובייקט ספציפי, לדוג' עבור משתנה סטטי "מונה אנשים" נגדיר מתודה סטטית "החזרת המונה".
ישנם מקרים שבהן יש לנו אוסף של מתודות שאינן פועלות על אובייקט ספציפי ולכן אנו רוצים לארוז אותם תחת מחלקה מסוימת כדוגמת מחלקת המתמטיקה שהינה מחלקה המכילה רק מתודות סטטיות.

API

Application ***P***rogramming ***I***nterface – 'שער' הכניסה לקוד. אילו מתודות הקוד שלנו מאפשר לשימוש חיצוני.
Minimal API מספק למשתמש את המינימום המידע הנדרש בשביל לדעת לתפעל את התוכנה .
יתרון המינימליות:

1. מעט מידע הופך את התוכנה למובנת יותר.
2. כל מה שכתוב ב API הוא בעצם מידע שהמשתמשים תלויים ומשתמשים בו ולכן קשה לשנות אותו לאחר מכן.

information hiding

Information Hiding – עיקרון המאפשר להגיע ל- minimal API
ג'אוה מאפשרת להגדיר <u>כל member</u> (מתודה או דאטא ממברס) בתור public או private.
Modifiers:
public – ה-data members והמתודות נראות לכל אובייקט מכל מחלקה אחרת .
private – ה-members גלויים רק לאובייקטים מאותה מחלקה.
לכן בהגדרת ממברים מסוימים להיות private נקבע המינימל API שהרי מה שייכנס ל API הוא רק מה שמוגדר כ public.
מה כדאי להגדיר כ-public ומה כ-private?
לרוב משתני מידע יש להגדיר כprivate. לעומת זאת במתודות זה תלוי החלטה.

Getters and Setters

על מנת לשנות ולגשת למשתנים private ניתן להשתמש ב -getters ו setters .
שימוש בגטר וסטר מאפשר לגשת ו/או לשנות private members ובנוסף ניתן להוסיף בדיקות מסוימות בתוך המתודות האלו.
כאמור, אנחנו לא רוצים לשתף את אופן המימוש שלנו בפונקציות מסוימות וזאת משום שהוא עלול להשתנות ולכן החבאת המידע יתאפיין גם במתן שמות כלליים בפונקציות public. דוג' נוספת היא לא לציין בשם של המתודה את מבני הנתונים שבו אנו משתמשים – בקיצור **לתאר מה עושים ולא איך עושים.**
ישנו מנגנון של גישה לפרייוט. כלומר פרייוט לא משמש עבור החבאת מידע אלא בשביל דזיין טוב יותר.

Encapsulation

איגוד רעיונות לכלל אחד המאפשר בין היתר את עיקרון ה-information hiding.

Method overloading

על מנת ליצור מתודות בעלות שם זהה עליהן להיות שונות ב:

1. כמות המשתנים.
2. ה-type של המשתנים.
3. סדר המשתנים.

הרצאה 3

נושאי ההרצאה:

1. Single Responsibility Principle
2. Inheritance.
3. Polymorphism.

עקרון האחריות היחידה - Single Responsibility Principle

דרישות בהן מחלקה צריכה לעמוד :

1. ניתנת לתיאור במילים פשוטות.
2. צריכה להכיל פונקציונאליות נוספת (כזו שלא ניתן לקבל מאובייקט שהוגדר לפני כן).
3. צריכה להיות רעיון כללי המתפרט למופעים שונים.

עקרון האחדות היחידה - השאיפה ביצירת מחלקה היא שמחלקה תענה על תפקיד יחיד או תייצג רעיון יחיד וכל ה-members שלה ישמשו אותו.

מחלקה בעלת יותר מתפקיד אחד היא בעלת סיכוי גבוה לקבלת שינויים - ושינויים אינם דבר חיובי בתכנות שהרי שינויים 1. מצריכים בדיקות מחודשות 2. מועדים לבאגים ולכן המטרה היא ליצור מחלקות האחראיות על תפקידים מצומצמים ככל הניתן.

לכן במקרה שבו אנו אכן נדרשים למספר רב של פעולות נעדיף לחלק את המשימה למספר קטן של פעולות תחת מחלקה כללית הנקראת manager. כך שבעת שינוי נוכל לשנות רק מתודה ספציפית - לא את השאר ולא את המנהל.

ירושה - Inheritance

סוגי יחסים בין מחלקות:

יחס Has - a - נקרא גם composition : משמעות יחס זה היא שאובייקט אחד 'שייך' לאובייקט אחר. לדוג': לאדם יש שם, לאופניים יש גלגלים.

מימוש/ייצוג: data members.

Is - a - מחלקה א' היא סוג של מחלקה ב'

לדוג' סטודנט הוא בן אדם. לכן סטודנט מכיל את כל המאפיינים (data members, methods) של בן אדם אך בן אדם לא מכיל את כל המאפיינים של סטודנט.

מימוש/ייצוג: ירושה - כאשר מחלקה מסוימת "מרחיבה - extends" מחלקת אב. A הוא subclass של b ו b הוא superclass של A. וזאת כאשר A יורשת את B מה שנקרא גם A extends B .

Instance-of - נראה דומה ליחס הקודם אך מייצג מופע ספציפי של מחלקה ולא ירושה שלה. לדוג' מאור הוא סטודנט ממש ולא מחלקה היורשת מסטודנט אך מאור הוא סטודנט היורש מאדם.

מימוש/ייצוג: יצירת instance.

תכונות ירושה

1. רקורסיבית - ניתן להמשיך את שרשרת התורשה.
2. טרנזיטיבית - ניתן לרשת את המחלקה האחרונה בשרשרת ובכך לרשת את כולם.
3. כל מחלקה יכולה להיות אב להרבה מחלקות אך יכולה לרשת רק **מחלקה אחת** – כלומר להיות בן רק של מחלקה אחת.

Object class - מחלקה כללית שכל המחלקות יורשות ממנה (גם בצורה נסתרת (implicit)). ויש לה שתי מתודות: האחת המרה ל-string והשנייה equal.

האובייקטים ה private (שדות, מתודות, קונסטרקטורים) אינם ניתנים לגישה מהמחלקות היורשות. לכן צריך להשתמש במתודות כלליות (getter and setter)

protected

Protected modifier - אלטרנטיבה ל- public and private . ה-protected מאפשר גישה למשתנים מסוג זה למחלקות היורשות בלבד. זה מעין private 'לכל העולם' ו public עבור המחלקות היורשות.
החיסרון ב protected - כאמור ב-API לא נמצאים private members אך ה protected members צריכים להיכנס ל API שכאמור הרחבת ה API הוא דבר שאיננו מעוניינים בו.
לכן נעדיף תמיד להגדיר private כשאפשר ובמקרים שבהם נצטרך ל protected נשתמש ב getters and setters במקום.

overriding

הרעיון הוא לרשת מחלקה **ולממש** אותה בצורה קצת שונה. התהליך פשוט, לוקחים שם של מתודה שירשנו אותה **וממשים** אותה בצורה אחרת.
קונספט ה super - מילה שמורה בג'אווה המאפשרת לגשת למתודה שנעשה לה overriding ממחלקת האבא. וזאת נעשה ע"י ()super.method . להבנתי: ה super הוא כאילו מחליף את ה this של האבא או בתור אובייקט ולכן עושים super. נקודה משהו.
בעת יצירת מחלקה היורשת ממחלקה מסוימת הקונסטרקטור הראשון שנקרא הוא **הקונסטרקטור הדיפולטיבי** של מחלקת האב. לדוג': כוון שכל סטודנט הוא גם בן אדם, יצירת סטודנט גם קוראת לקונסטרקטור של האבא בין באופן יזום ובין באופן אימפליסיטי ולכן על מחלקת האבא להכיל דיפולט קונסטרקטור וכמובן שמחלקת האבא היא זו שתפעל בתחילה.
בפועל: אם למחלקה אין קונסטרקטור דיפולטיבי עלינו לקרוא לקונסטרקטור הקיים באופן explicit אך אם יש לו דיפולט קונס' אין צורך לקרוא ל super באופן יזום כוון שזה ייקרא באופן implicit.
ירושה יכולה לשמש מנגנון למחזור קוד אך ישנן אלטרנטיבות טובות יותר כמו composition (ניגע בהמשך)

polymorphism

Polymorphism - העיקרון הבסיסי ביותר בתכנות מונחה עצמים (מיוונית: ריבוי צורות) - היכולת של אובייקט מסוים להיות מסוגים שונים - לקחת על עצמו כמה צורות. מה שבא לידי ביטוי בהרחבת מחלקות.
סוג הרפרנס מגדיר מה האובייקט **יכול** לעשות - מתודות, **ותוכן** הפעולה הוא במה שהוכנס לאובייקט (הפרה בדוגמא בהמשך).
לסיכום: **הכרת** המתודה תלויה ברפרנס ו**תוכן** המתודה הוא בסוג האובייקט.
לדוג': אם cow הוא רפרנס מסוג פרה ו animal הוא רפרנס מסוג חיה אזי animal = cow יגרום לכך ש-animal יכיר את המתודות השייכות למחלקת החיה בלבד אך תוכנן יהיה זה של הפרה.

פולימורפיזם מאפשר החלפת אובייקט (דוגמא של שינוי animal מפרה לכלב).
דוגמא נוספת של פונקציית מיון המקבלת numbers ויכולה לקבל אינט ודאבל.

shadowing

Shadowing - בשדות (שאינן מתודות) שבהן השמות זהים בין מחלקת האב למחלקה היורשת התוכן נקבע לפי ה reference type.
בנוסף, מתודות סטטיות הן גם נקבעות לפי ה reference type - ההיגיון בכך הוא שהמתודות הסטטיות הן קשורות למחלקה הכללית ולא לאובייקט ספציפי של המחלקה, ואנו מעוניינים בקבוצת הרפרנסים ולא בקבוצה של האובייקט הספציפי.
באופן כללי מומלץ לא להשתמש ב shadowing - זה יוצר קוד מבלבל (נשים לב שגם בשביל לאפשר shadowing צריך לא להשתמש ב private).

עקרון תכנות לממשק ולא למימוש - Program to interface, not to implementation

עקרון תכנות שבו ה- API לתאר את הרמה הכללית של המחלקה המורישה (מבחינת שמות מתודות וכדו').
יתרונות:

1. מאפשר שמתודה מסוימת תעבוד לכמה אובייקטים (כל מה שהוא תחת חיה).
2. מאפשר שינוי בקוד בצורה קלה יותר כוון שהמשתמש בתוכנה לא מתייחס לאובייקט ספציפי.
3. ומאפשר שימוש קל יותר בקוד ולמידה שלו שהרי מתרחשת למידה של אובייקט כללי יותר מעבר לאובייקטים הפרטיים.

לדוג': כאשר יוצרים מתודת השמעת קול לחיה נשתמש בשם makeSound ולא makeMoo.

הרצאה 4

מחלקות אבסטרקטיות

במקרה שבו יש לנו מחלקה כללית ומחלקות פרטיות הממשות מתודה כללית אך בצורה מעט שונה (כמו דיבור) נרצה ליצור ייחוד עבור אותה פעולה (כמו השמעת קול). ונראה פתרונות לבעיה זו :

פתרון 1 - לוותר על המתודה הזו במחלקה הכללית. אך זה יוצר כמה בעיות:

- הרחבת הקוד לשוווא.
- נוצר API שונה לכל חיה.
- לא ניתן ליצור פולימורפיזם כוון שלא נוכל להשמיע קול לחיה שהרי אין לה את המתודה הזו.

פתרון 2 - מימוש ריק ודריסת המתודה בכל מחלקה פרטית והחסרונות הן:

- במידה ושכחנו ליצור מימוש - תהיה פעולה ריקה.

פתרון 3 - מחלקות אבסטרקטיות.

מחלקה אבסטרקטית היא מחלקה שלא ניתן ליצור לה instance ומטרתה להיות בסיס למחלקות שירשו אותה- כלומר לא ניתן ליצור new.

יצירת מחלקה אבס' היא ע"י המילה השמורה abstract .

Abstract Methods - למחלקה אבסטרקטית יכולות להיות מתודות אבס' ומתודות שאינן אבס'. המתודות שאינן אבס' ניתנות למימוש במחלקה האבסטרקטית והמתודות האבסטרקטיות חייבות להיות ממומשות במחלקות היורשות אותו. - יצירת מתודה אבסטרקטית בעצם מאלצת את המחלקה היורשת שלה לממש אותה. כך שבנוסף לרווח של הפולימורפיזם ישנו הרווח של מניעת שגיאות מתכנת - בכך שאני מאלץ את הקוד לא לעבור קומפילציה. מעין קונסט.

מאפיינים:

1. מחלקה אבס' יכולה לרשת מח' אבס' אחרת. אך אינה חייבת לממש את המתודות האבס' שלה.
2. כל מה שמח' רגילה יכולה לעשות גם מח' אבס' יכולה לעשות וההבדל היחיד הוא שהיא לא יכולה ליצור אינסטנס.
3. מתו' סטטית לא יכולה להיות אבס' (לא נרחיב).

מה לא ניתן לעשות:

1. לא ניתן לקרוא למתודה של מחלקת האב (האבס') ע"י super .
2. לא ניתן ליצור מחלקה אבס' בתור private - שהרי ה private שייך לאינסטנס מסוים ולמח' אבס' אין אינסנטס.

הסיבות לשימוש במח' אבס':

1. כאשר אין מופע של המח' באופן האבס' שלה. כמו המושג חיה, אין חיה שהיא מוגדרת כחיה בלי מופע ספציפי.
2. כאשר אנו רוצים להכריח API מסוים.

interfaces

דומה למחלקה אבסטרקטית ויכול להכיל רק שני דברים - קבועים ומתודות אבסטרקטיות.

לא ניתן ליצור אינסטנס מאינטרפייס אך ניתן לממש אותם באמצעות מחלקות או לרשת מהם ע"י אינטרפייסים אחרים.

- ניתן לממש אינטרפייס ע"י מחלקה ושימוש במילה implements.

האינטרפייס אינו מגדיר אובייקט בעולם אלא מייצג רשימת תכונות שהמחלקה לוקחת על עצמה. הסיומת המתאימה לאינטר' היא able. לוקחת על עצמה הכוונה שבתוך המחלקה יש את האפשרות לעשות משהו.

מאפיינים:

1. האינטר' לא מדבר על המימוש אלא רק על ה"המה" - מה המחלקה יכולה לעשות.
2. אינטר' לא מגדירים מחלקות שאינן public - וזה קשור להיותם מגדירים API.
3. אינטרפייס אינו כולל data members מלבד קבועים (final static) וכן לא כולל מתודות שאינן public (מעצם היותו מגדיר api)
4. ניתן להרחיב את האינטרפייס והמחלקה המממשת את הסאב-אינטרפייס צריכה כמובן לממש את המתודות הנמצאות בשני האינטר', הסאב והסופר.
5. המחלקה המתחייבת לאינטרפייס מסוים חייבת לעמוד בדרישות שלו וכמובן עם אפשרות להרחבה.

החוזה יכול לדרוש דרישות קדם אך מחלקה יכולה לומר שהיא דורשת פחות דרישות קדם בשביל לעמוד בחוזה.

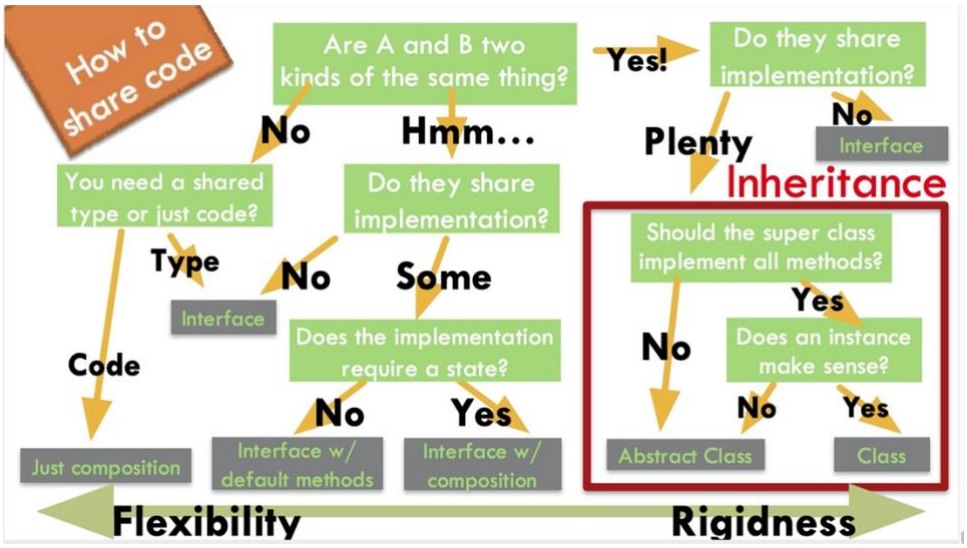
אך היא לא יכולה לדרוש יותר מזה ובנוסף היא יכולה לממש יותר ממה שדרוש באינטרפייס . (האינטרפייס הוא כמו המעסיק באנלוגיה של מעסיק ומועסק).

6. כל מחלקה יכולה לממש מספר רב של אינטר' לעומת זה שהיא יכולה להרחיב רק מחלקה אחת.

ניתן להסתכל על מחלקה בכמה types - 1. הטייפ שלה 2. אלו שהיא יורשת מהם 3.האינטר' שהיא מממשת.
מח' אבס' ואינטר' הם די דומים זה לזה מבחינת אי היכולת ליצור אינס' ומבחינת יצירת היררכיית מחלקות וכפיית API על המשתמשים שלה.
כללי אצבע לשימוש בכל אחת מהן :

- אם א' הוא **סוג של** ב' אז עדיף להשתמש בירושת מחלקה.
- אם א' הוא סוג של הסכם או משהו שא' רוצה לממש מב' אז עדיף להשתמש באינטר', כמו מסמך ותכונת הדפסה.
- כאשר אין דרך טובה להחליט עדיף לבחור באינטר' ומצד הסיבה הפשוטה שניתן לממש כמה אינטר' וזה לא מגביל את השימוש כמו במחלקה אחת.

עץ החלטה בבחירת מחלקה או אינטרפייס



הרצאה 5

reuse mechanisms

אפשרויות מחזור קוד:

- Inheritance
- Composition - הכלה

בקומפוזישן אנחנו בעצם "מכילים" אובייקט אחד באובייקט אחר (ע" הגדרתו כ data member לדוג') ומבצעים עליו את הפעולה שאנו רוצים (במקום לכתוב אותה שוב).
קומפוזישן נקרא black box כוון שאנו לא יודעים מה יש בתוך הפונקציה הזו אלא רק מה התוצאות שלה, לעומת ירושה שנקראת white box שבה יש לנו גישה לפונקציות ה public and protected שלה.

חסרונות ויתרונות:

Inheritance – ירושה	
יתרונות	**חסרונות**
• קלה לשימוש - נתמכת ע"י שפת הקוד. • מאפשרת פולימורפיזם. • מוגדרת באופן סטטי בזמן קומפילציה (בעל יתרונות וחסרונות). • קל יותר לשנות את המימוש ע"י overriding או שימוש ב super.	• בלתי ניתן לשינוי במהלך זמן ריצה. • שבירת ה - encapsulation ע"י חשיפת המידע הפנימי של המחלקה המורישה אל המחלקה היורשת. • המחלקה מחויבת למימוש של המחלקה מורישה - חסרון במקרה שבו המחלקה המורישה מגדירה פונקציות נוספות שהמחלקה היורשת אינה מעוניינת בהם. • ניתן לרשת רק מחלקה אחת.
קומפוזישן - composition	
יתרונות	**חסרונות**
• מוגדר בזמן ריצה - ניתן לשנות את המחלקה שאנו מכילים במהלך זמן הריצה. • ה encapsulation אינו נשבר - שהרי ה api אינו מתגלה. • לא קיימת תלות בין האובייקטים השונים שהרי אנחנו רק מחזיקים אותו ויכולים לשחרר אותו כשנרצה. • התמקדות בפעולה אחת. • הכלת מספר רב של מחלקות.	• פולימורפיזם אינו נתמך. • מחלקה דינמית - פוטנציאל לטעויות (שכחה וכו'). • מבנה מורכב יותר - סיכוי גבוה יותר לבאגים.

לסיכום – עצות לתעדוף בין ירושה להכלה:

- אם אנו מעוניינים בעיקר במחזור קוד - קומפוזישן.
- במקרה שבו אובייקט A הוא סוג של B - ירושה.
- וכאשר יש שתי מחלקות ש A יכול להיות שייך להם - נבחר את המחלקה ש A הוא יותר "סוג שלה" ובמחלקה השנייה נשתמש בקומפוזשין.
- כאשר אנו מעוניינים בפולימורפיזם אך שני האובייקטים <u>אינם</u> "סוג של" - נמליץ להשתמש באינטרפייס.

Casting

Casting - קסטינג הוא הלב של פולימורפיזם - משמעותו לקחת אובייקט מסוג אחד ולהסתכל עליו בעיניים של אובייקט אחר.

סוגי קסטינג:

up casting - סוג זה הוא מצב שבו ה - reference type משמאל הוא super class של האובייקט מימין. ובעצם באובייקט שאליו משימים מכניסים אובייקט מדרגה נמוכה ובכך "מעלים" אותו.

ישנם שני סוגי קסטינג:

- Implicit casting - הקומפיילר מחליט מה סוג האובייקט לדוג': Animal myAnim = new Dog().
- Explicit casting – המרה גלויה, נעשה ע"י (type) לפני האובייקט מימין לדוג': myAnim = (Animal) myDog

Down-casting - המרת אובייקט ל תת-אובייקט שלו. כמו המרת חיה לפרה : Cow myCow = (Cow) new Animal();

- חייב להיות מפורש.
- דאון קסטינג יעבוד רק כאשר הטייפ ה"אמתי" של האובייקט הוא האובייקט שאליו אנו רוצים להעביר (כלומר כאשר עשינו אפ קאסטינג לאובייקט מסוים ואנו מחזירים אותו).

באופן כללי, לא מומלץ להשתמש ב down casting. כוון שכשל של dc מתגלה בעיקר בזמן ריצה וזה לרוב דבר שלא המתכנת מגלה כי אם הלקוחות שמשתמשים בתוכנה שלו.

Instance-of - אופרטור המאפשר בדיקה האם אובייקט מסוים הוא אינסטנס של אובייקט אחר מה שיכול עזור במקרה של dc .

ככלל לא מומלץ להשתמש בו.

חסרונות:

- מועד לבאגים - בדיקות נוספות שצריך לעשות שניתן ליפול בהם.
- יוצר חוסר גמישות - שהרי הגמישות שנובעת מה up casting היא הכלליות שבו ולכן קוד אחד מתאים לכמה סיטואציות. ולכן כל ירידה לפרטים עלולה לקבל שינוי.

אם זאת, למה לומדים זאת? כוון שיש המשתמשים בבך וכדאי להכיר.

Design patterns

Design patterns – תבניות עיצוב - רעיון הנובע מעולם הארכיטקטורה ומשמעותו מתן מבנה כללי ופרקטי לבעיות שחוזרות על עצמן.

דייזין פטרן מציע:

1. גישת פתרון כללי לבעיה ספציפית.
2. פתרון אלגנטי שניתן בקלות להרחבה.
3. מאפשר לצפות בעיות שעלולות לצוף בזמן העיצוב.
4. אינו תלוי תוכן או שפת קוד.

דוגמא:

delegation design pattern - יצירת מחלקה מסוימת בעלת API זהה לזו של מחלקה אחרת מבלי להשתמש בירושה (ת'כלס קומפוזישן שמשותף בכל או רוב המתודות (api זהה))

יתרונות:

1. מאפשר שינוי מימוש בזמן ריצה.
2. המחלקה המרכיבה מאפשרת ירושה ממחלקה נוספת.

מאפייני design patterns:

1. <u>Design pattern name</u> - מאפשר חלוקת קוד ושפה משותפת בין מתכנתים.
2. <u>Problem</u> - ה-design pattern צריך להגדיר את הבעיה ומרכיבי הבעיה עליה הוא מעוניין לענות.
   מה הבעיה? מדוע נעדיף להשתמש בעיצוב הזה? מה התנאים המקדימים לכך שנעדיף להשתמש בעיצוב הזה?
3. <u>Solution</u> - מרכיבי הפתרון והיחסים ביניהם, פתרון כללי.
4. <u>השלכות</u> - יתרונות/חסרונות, זמן ריצה, יכולת הרחבה וכו'.

סוגי עיצוב:

1. **Creational patterns** - מתעסק ביצירת אובייקטים - instantiation - לדוג': factory .
2. **Structural pattern** - מתעסק במבנה של האובייקט (קומפוזישן). לדוג': facade , delegation.
3. **Behavioral patterns** - התנהגות האובייקט או הקשרים בין האובייקטים. לדוג' איטרטור.

Facade

שיטת עיצוב זה שייכת ל structural pattern.

Facade (פנים בצרפתית) מגיע מעולם הארכיטקטורה - משמעותו לקיחת מערכת מורכבת והתבוננות בה דרך מבט צר.

Facade עוסק בבעיות שבהן המערכת מורכבת מיחסים וקשרים מסובכים בין רכיבי התוכנית מה שיוצר api מורכב אך המשתמש צריך api צר יותר. כלומר, המטרה היא ליצור את אותה תוצאה אך להקל על המשתמש את הבנת ה api.

דרך הפעולה:

1. בניית מחלקת facade .
2. מימוש ה facade בהתאם ל api המצומצם הדרוש.
3. שימוש ב-api המסובך במחלקת ה facade עם שמירה על ה api המצומצם משלב 2.

יתרונות:

1. מצד הלקוח - שימוש פשוט.
2. כל שינוי במערכת המורכבת של הapi המורכב אינו גלוי למשתמש.
3. ה api המורכב לא נאבד, משתמש הרוצה להשתמש ב api המורכב יכול לעשות זאת.

נשים לב כי facade <u>אינו מוסיף</u> יכולות חדשות אלא רק <u>מארגן ומסדר את הקיימות</u> בצורה פשוטה ומובנת.

תרגול 5

Open Hashing – שיטת גיבוב שבה איברים הממופים לאותו תא בטבלת הגיבוב מסודרים בתוך רשימה מקושרת.

Closed hashing – שיטת גיבוב שבה כאשר איבר ממופה לאיבר 'תפוס' מדלג (באמצעות נוסחה מסוימת) למיקום הפנוי הבא. (ראינו בתרגיל 4 כי הנוסחה היעילה ביותר היא : $\frac{(i+i^2)}{2}$).

מתודת hashCode() - על מנת שאובייקט יוכל להיות ממופה בתוך טבלת גיבוב, עליו לממש את המתודה hashCode().

באופן דיפולטיבי פונקציית ה-hash יוצרת hashing על המספר בזיכרון אך כמובן שניתן לדרוס ולממש לפי הצורך, לדוג' באובייקט מסוג String, הגיבוב נוצר על תוכן המחרוזת.

חשוב לשים לב שהערך שעליו מתבצע ה-hashing (data member כלשהו) חייב להיות קבוע - final.

הרצאה 6

Collection

Collection - אובייקט המכיל אובייקטים אחרים - ת'כלס מבני נתונים.
במב"נ אנו מאחסנים מידע מסוים, מפעילים עליו פעולות מסוימות וכו' שלרוב מייצגים אובייקט הקיים בעולם.
Collection framework - ספריה המכילה מבני נתונים.
ספריה זו מכילה:

1. Interfaces - מבני נתונים אבסטרקטים המאפשרים מימושים שונים כמו List, Map וכדו'.
2. Implementations - מימוש ספציפי של מב"נ כמו LinkedList, HashMap, HashSet וכדו'.
3. Algorithms - פעולות שניתן לבצע על האלגוריתמים האלו.

על מנת להשתמש יש לייבא אותם ע"י import.java.util*.
חשיבות הספריה:

- חוסך זמן עבודה.
- בכל הנראה מבנה זה מיוצר בצורה איכותית יותר מאשר נכתוב אותה בעצמנו.
- השימוש באותו מבני נתונים בין משתמשים שונים יוצר ממשק נוח בין המשתמשים.

הקדמה ל Generics

Generic Class אינו עוסק בהפשטה והפרטה של אובייקטים כמו שעסקנו עד עכשיו (כמו ירושה) אלא בסוג של האובייקטים בהם אנו עוסקים.
ביצירת אובייקט של generic class אנו יוצרים אובייקט עם טייפ ספציפי - לכן לאחר יצירת האובייקט לא נוכל להכניס טייפ שונה לאובייקט של המחלקה הגנרית - אם כן הגנריות מתאפיינת רק ביצירת האובייקט.
כל המרכיבים ב collections הם גנריים. לדוג' קיימת המחלקה: public class LinkedList<E> .

Collection interfaces

7 אינטרפייסים בסיסיים:

1. Collection
   set, sorted set, list, queue
2. Map
   Sorted map

List - אוסף מסודר (לא בהכרח ממוין).
מאפשר:

- גישה לפי אינדקס.
- חזרה של אברים.

Queue - אוסף של איברים המאפשר גישה לאיבר אחד באוסף - ראש התור.
הסדר שבתור נקבע לפי :

- FIFO - הראשון נכנס הוא הראשון לצאת.
- סדר עדיפויות - כמו תור לניתוח מוח.

בכל מקרה יש להגדיר את ראש התור בצורה ברורה.
מאפשר:

- הכנסה - push
- גישה לראש התור.
- הוצאת ראש התור.

Set - אוסף שאינו מאפשר חזרת איברים.
Sorted Set
Map - אובייקט הקושר בין key ל value .
מפתח הוא יחיד - לעומת ערך שאינו בהכרח יחיד.

אנלוגי ל set ברמת המפתחות.
Sorted Map - מיפוי שבו המפתחות מסודרים.

implements

על פי קונבנציה הוגדרו שני קונסטרקטורים שכל מחלקה בספרית הקולקשיין צריכה לממש:
Default constructor - יוצר אובייקט ומאתחל את כל איבריו באופן ריק
Copy constructor - קונסטרקטור המקבל קולקשיין אחר ומכניס את כל האיברים שבתוך הקלט אל אובייקט הנוכחי.
זוהי קונבנציה - כלומר ניתן לממש ולקמפל גם בלי שני אלו אך על מנת לשמור על השפה המשותפת יש לעבוד לפי הקונבנציה.
אינטרפייסים והמימושים שלהם:

1. List
   - ArrayList - מערך בעל גודל משתנה.
     מגדיר get ו set בזמן קבוע.
     Index of, remove לוקח o(n) .
     add - הוספת n איברים תארך o(n) אך הוספת איבר אחד יכולה לקחת יותר מ o(1).
   - Linked List - רשימה מקושרת
     Add -זמן קבוע
     Get, set, index of, remove - o(n) .
     תופסת מקום קטן יותר בזיכרון המחשב מאשר ArrayList .
2. Set
   - Tree Set - קבוצה המבוססת על מבנה של <u>עץ Avl</u> <u>ממויין.</u>
     Add, remove, contains - o(logn).
   - Hash Set - קבוצה המבוססת על hash table.
     Add, remove, contains - בממוצע זמן קבוע.
3. Map
   Tree Map
   Hash Set

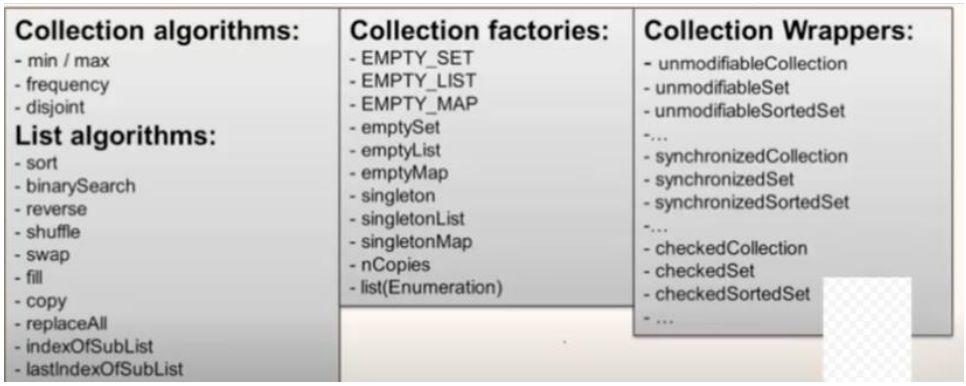
סיכום:

Computational Complexity

	Add	Remove	Get by index	Contains	Iteration
ArrayList	O(1)*	O(N)	O(1)	O(N)	O(N)
LinkedList	O(1)	O(N)	O(N)	O(N)	O(N)
HashSet	O(1) avg	O(1) avg	–	O(1) avg	O(T)**
***TreeSet**	O(logN)	O(logN)	–	O(logN)	O(N)

* *amortized constant time*, that is, adding *n* elements requires O(*n*) time
** see later

פעולות מרכזיות שקיימות במבני הנתונים:



Hash tables

טבלאות גיבוב נוחות עבור מקרי שבהם אנו רוצים לבצע חיפוש איברים ומחיקתם בצורה היעילה ביותר.
טבלת גיבוב משתמשת בפונקציית גיבוב - פונקציה דטרמיניסטית הלוקחת key וממירה אותו למספר בטווח מסוים (גודל המערך אליו ממפים).
במקרה של התנגשות - ישנם מספר פתרונות למיתון הבעיה.
איטרציה על טבלת גיבוב היא בעצם מעבר על גודל הטבלה שגודלה גדול או שווה למספר האיברים הקיימים בפועל ברשימה. לכן במקרה בו נרצה לבצע איטרציה באופן יעיל נעדיף להשתמש ב treehash .

Iterators

איטרטור הוא אובייקט המקבל מבני נתונים ועובר על כל איבריו.
לאיטרטור ישנם שתי מתודות:

1. HasNext() – מחזיר true אם המב"נ אינו ריק.
2. Next() – מקדם את האיטרטור לאיבר הבא ומחזיר את האובייקט <u>עליו האיטרטור מצביע</u>.
   כלומר עבור איטרטור המצביע על האיבר הראשון, ביצוע פעולת הnext תחזיר את האובייקט הראשון ותקדם את האיטרטור להצביע על האובייקט השני.

איטרטור הוא גם כן גנרי - ועליו להיות תואם למבני נתונים עליו הוא רץ.
חשיבות האיטרטורים:

1. יצירת הפרדה בין מבני הנתונים לבין האופן שבו אני עובר עליו - מאפשר חופש בבחירת מבני הנתונים.
2. מאפשר חופש באופן המעבר על האיברים, מהסוף וכו'.
3. איטרטורים לרוב יותר יעילים.
4. מאפשר גם למחוק איברים תוך כדי ריצה ברשימה מקושרת לדוג'.

ניתן להפוך אובייקט להיות איטרבילי על ידי מימוש ה אינטרפייס iterable שעל מנת לממשו יש לממש את המתודה :
public Iterator<> iterator() { return new Iterator;}
ועל מנת לממש את הinterface של ה- iterator הזה יש לדרוס את המתודות:

1. Public boolean hasNext() {\\code}
2. Public <iteratorType> next(){\\code} - המתודה הזו מחזירה את האובייקט הנוכחי ומקדמת את האיטרטור להצביע על האיבר הבא.
3. Public void remove(){\\code}

הרצאה 7

הקדמה ל exceptions

סוגי השגיאות:

1. שגיאות קומפילציה - שגיאות נוחות כוון שהמתכנת מקבל התראה ע" הקומפיילר.
2. שגיאות זמן ריצה - שגיאות שאינן מתקבלות מהקומפיילר, והן דורשות טיפול ספציפי של המתכנת והן תוצאה של באגים או קלט שגוי וכדו'.

תכנית טובה:

- יודעת 'להתאושש' משגיאות כראוי ובזמן המתאים.
- יש לציין את הטיפול בשגיאות ב documentation.

Exception -' הודעה' המתריעה שהתרחש אירוע לא תקין.

- אלטרנטיבי לערכי החזרה (שגיאה -1 וכו')
- דרך הפעולה: ברגע שמתרחשת שגיאה, המתודה "זורקת" שגיאה.
- השגיאה הנזרקת היא אובייקט בג'אווה.
- הטיפול בשגיאה יכול להיעשות במתודה שבה נזרקה השגיאה או במקרה שבו אין היא יודעת לטפל בשגיאה זו, השגיאה נזרקת הלאה למתודה שקראה לה.
- בכל מתודה ניתן "להוסיף" אופציית exception ע"י המילה השמורה throws (כמו שכותבים בהרחבת מחלקה). לדוג': public int getIndex(int i) throws ListException
- הוספת throws נוספת ל API ומודיעה כי קיים פוטנציאל לשגיאה - לכן חשוב להוסיף ב javadoc איזה exceptions יכולים להיזרק ומתי.
- ברגע שמגיעים ל exception - המתודה עוצרת. הן מבחינת ששום קוד נוסף לא ממשיך לרוץ והן מבחינת ערכי החזרה בהמשך הקוד שלא יוחזרו.

טיפול ב exceptions:

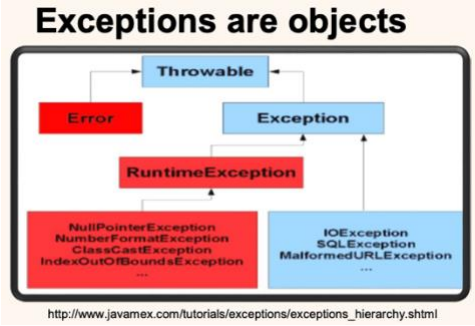
- נעשה ע"י try ו catch. בתוך ה-try יופיע קטע הקוד שאנו רוצים להריץ. וב-catch יופיע קטע הקוד שינהל את השגיאה.
  סינטקס :

```
try {
        //get index from user
        int element = list.get(0);
        //...
} catch(ListException e) {
        // handle list errors
} catch(OtherException e) {
        // handle other errors
}
// Rest of program
```

- בתוך ה catch נתפס אובייקט ממש לדוג' ListException e - המשתנה e הוא אובייקט מסוג ListException. האובייקט הזה מכיל מידע רלוונטי בנוגע לשגיאה (מעבר כמובן לעצם ההבחנה הפשוטה איזו סוג שגיאה זו).
- מתודה שקיבלה exception אך אינה יודעת לטפל בו יכולה לזרוק אותו אחורה למתודה שקראה לה. - במתודה זו לא יהיה try/catch אלא פשוט נממש את המילה השמורה throws בחתימת הפונקציה והמתודה כבר תעביר את השגיאה הלאה.
- ניתן לתפוס שתי סוגי שגיאות בcatch אחד באמצעות הסינטקס: catch(Exception1 | Exception2 e)

exception types

אילן ירושות של exceptions :



שגיאות error הן לרוב שגיאות של מערכת ההפעלה שלרוב לא נתעסק בהן ברמת הנדסת התוכנה.

Unchecked error - לרוב נובעות מבאגים בתוכנה שתלויים במתכנת - ולכן אנחנו לא מצפים שהם יקרו באופן מפתיע.

- יורשת מהמחלקה RunTimeException.
- יכול להתרחש בכל תכנית ובהרבה תרחישים.
- אין צורך להשתמש ב try-catch או לנהל את השגיאה במתודה הקוראת.

Checked errors - שגיאות הנובעות מהמשתמש.

- יורשת מהמחלקה exceptions.
- לרוב ספציפי לתוכנית מסוימת.
- הכרחי לבצע בשגיאה זו :
  - לממש throws במתודה שבה מתרחשת השגיאה.
  - על השגיאה להיות מטופלת באחת המתודות שקראו לה – try-catch.
  - אי עשיית הנ"ל תוביל לשגיאת קומפילציה.

הסיבות לשימוש ב exceptions

1. הפרדה בין הקוד שמבצע פעולה מסוימת לבין הקוד שמנהל את השגיאות.
2. פעפוע שגיאות במעלה הקוד - טיפול בשגיאות במסודר במקום מרוכז ולא בהכרח במקום בו צפה השגיאה.
3. איחוד/הבחנה בין שגיאות - מאפשר טיפול דקדקני בכל סוג של שגיאה או טיפול באופן כללי בסוג שגיאה מסוים - על פי עיקרון הפולימורפיזם.

אין להשתמש במנגנון ה try-catch להחלפת הפעולות הלוגיות הרגילות כמו תנאי if else וכדו' מכמה סיבות:

1. חוסר יעילות - שימוש במנגנון הזה מאט את התוכנה.
2. לא צפוי ומחביא באגים - אם התוכנה זרקה בטעות exception מסוים שלא צפינו לו אנו נחשוב שהשגיאה שעלתה היא זו שצפינו לה ובכך נסיים את התוכנית לפני שהיא הושלמה.

מקרים לא נפוצים אך שימושיים במנגנון הexception:

1. חזרה מהירה במעלה הקוד.
2. Timeout - במקרה ואנו רוצים להריץ קוד למשך זמן מסוים, בג'אווה משתמשים ב exception בשביל "לעצור את הזמן" - לא נעסוק בקורס שלנו.
3. מתודה אופציונלית - כל מימוש רשאי להחליט האם הוא רוצה לממש את אחת מהאופציות האלו.
   - ה documentation אחראי לתיאור אילו מתודות אופציונאלית נתמכות.

Packages

דרך לחלוקת הקוד לחלוקה הגיונית מסוימת - בדומה לחלוקת תיקיות במחשב.

יתרונות:

1. מארגן ומחלק את הקוד לקבוצה של מחלקות ואינטרפייסים הקשורים זה לזה.
2. חלוקת הקוד למודולים עצמאיים.
3. מאפשר הרשאות מסוימות לקוד .

- collection framework - Java.util הוא חבילה לדוגמא.
- על מנת להגדיר חבילה יש להשתמש במילה השמורה package.
- השם נכתב באותיות קטנות.
- במקרה שבו מחלקות מוגדרות בחבילות שונות יש לייבא את החבילה. Import pack.<class Name>

הרשאות:

- מחלקות באותה החבילה חשופות ל protected אחת של השנייה.
- בחבילה יכולות להיות מחלקות שהן public - המופיעות ב api וחשופות לשימוש. ומחלקות אחרות (ללא modifier) שאינן גלויות מלבד למחלקות שבתוך החבילה עצמה.

סיכום הרשאות גישה:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default(=**package**)	**Y**	**Y**	**N**	**N**
private	Y	N	N	N

- Exception וחבילה - מומלץ לשים את כל השגיאות בחבילה המתאימה להם ולא לממש חבילה של שגיאות. וזה מתקשר לרעיון של חבילה של איגוד מחלקות תחת כותרת אחת - ושגיאות של מחלקה קשורות יותר לחבילה עצמה מאשר לחבילות אחרות.

Nested classes

המחלקה החיצונית מתוארת כ: enclosing/wrapping/outer class .

הסיבות לשימוש:

1. קיבוץ לוגי של מחלקות - במקרה שבו מחלקה מסוימת מיושמת רק במחלקה אחת אז מומלץ ליצור קינון.
2. העצמת הencapsulation
   - המתודה הפנימית יכולה לגשת ל- private members של המחלקה שמעליה.
   - המחלקה פנימית עצמה יכולה להיות מוגדרת כ private.
3. קוד מובן יותר - הקוד אולי נראה מסורבל יותר כוון שיש פה שתי יחידות אחת בתוך השנייה, אך המתבונן בקוד מבין מיד שעצם קיומו של הקוד תלוי במחלקה העוטפת אותה.
4. מתי להשתמש:
   - במחלקה פנימית קטנה.
   - מחלקה פנימית private .
5. הרשאות (של המחלקה החיצונית לפנימית):
   - המחלקה הפנימית יכולה לקבל: public, package ,private. Protected .
   - מחלקה חיצונית יכולה לגשת לשדות פרטיים של מחלקה פנימית רק עם אינסטנס של הפנימית.
   - מחלקה חיצונית יכולה לגשת לשדות סטטיים של מחלקה פנימית סטטית אך לא יכולה לגשת לשדות שאינם סטטיים.

6. סוגי מחלקות פנימיות בג'אווה:

Static - מחלקה זו אינה שייכת לשום אינסטנט של המחלקה העוטפת - מעין מחלקה העומדת לעצמה אך מוכנסת לתוך מחלקה אחרת מצד ענייני packaging. ולכן היא נוצרת ללא תלות במחלקה העוטפת.

הרשאות:

- יכולה לגשת לכל ה- data members אך ורק דרך אובייקט - ולא נדרשת לאינסטנס ספציפי. כך שהגישה אליה היא דרך השם של המחלקה העוטפת : enclosingClass.staticClass.fieldOrMethod.
- יכולה לגשת למשתני private של העוטפת - כלומר בעת יצירת אובייקט של המחלקה העוטפת (לדוג') בתוך המחלקה המקוננת, האובייקט של המחלקה העוטפת יכול לגשת למשתני הפרייוט שלו.

Inner – מחלקה פנימית שאינה סטטית המשויכת לאינסטנס ספציפי של המחלקה העוטפת אותה. יכולה להיווצר רק ע"י יצירת אינסטנס של המחלקה העוטפת. כלומר:

```
OuterClass outerInstance = new OuterClass();
OuterClass.InnerClass inner = outerInstance.new InnerClass();
```

- הסוג הנפוץ ביותר member class - מחלקה מקוננת שהינה גם member של המחלקה העוטפת.

הרשאות:

- מחלקה פנימית יכולה לגשת לכל השדות של זו החיצונית (ללא אינסטנס של החיצונית).
- כוון שהיא שייכת לאינסטנס ספציפי - היא אינה יכולה להכיל משתני static.

הערה: לרוב לא נרצה להגדיר exceptions בתור מחלקה מקוננת כוון שהexceptions צריכים להיות גלויים ב api ובמחלקות מקוננות אנו שואפים שהמחלקה הפנימית לא תהיה גלויה אלא תשמור על ה private.

Closure, Local class, Anonymous

Local class - סוג של מחלקה מקוננת המוגדרת בתוך מתודה.

- מחלקה לוקאלית נגישה בתוך אותו סקופ.
- רלוונטית רק בקונטקסט של אותה מתודה - ולכן מתאים לשימוש רק במקרה כזה.

מאפיינים:

- אין משמעות להגדיר modifier למחלקה פנימית בתוך מתודה.
- אינה יכולה להיות static שהרי היא קשורה לאינסטנס ספציפי.
- אינה יכולה להיות אינטרפייס.

הרשאות:

- יכולה לגשת לכל ה data members and methods של האינסטנס של המחלקה שבה המתודה נמצאת.
- יכולה לגשת רק למשתני ה final של המתודה העוטפת אותה.

Anonymous class - מחלקה לוקאלית ללא שם.

- מחלקה שמשתמשים בה פעם אחת.
- ניתן להגדיר בתוך כל מקום.
- ניתן להגדיר על מחלקה קיימת (ע"י override) או ע"י אינטרפייס (שבכך נוצרת מחלקה הממשת את האינטרפייס הזה) – כמו במקרה של comparator.
- דוגמת סינטקס:

```
String[] filelist = dir.list(
        new FilenameFilter() {  // Creating an instance while
                                // implementing the class
        public boolean accept(File dir, String s) {
            return s.endsWith(".java");
        }
    } // end of class declaration
); // end of the statement of calling dir.list()
```

Note that the statement ends with a semi-colon

Closure - בלוק של קוד (לרוב פונקציה) בעל קשר לא טריוויאלי עם משתנים מחוץ לו.

- לרוב ממומש באמצעות פונקציות מקוננות.

- במובן מסוים מחלקה לוקאלית היא קלוסר.
- לעומת מחלקה לוקאלית, קלוסר יכול לגשת למשתנה שאינו סטאטי.

הרצאה 8

חלק 1 - מודולריות

Modularity – פירוק התוכנה ליחידות שאינן תלויות אחת בשניה.

יתרונות:

1. קל לתחזוק.
2. מאפשר לקחת בעיה מורכבת ולפרק אותה לתתי בעיות.
3. מאפשר עבודה מחולקת בין צוותי עבודה שונים - בנפרד ובאופן בלתי תלוי.

עקרונות עיקריים:

1. **פריקות Decomposability** - המטרה לקחת בעיה ולפרק אותה לתתי בעיות קטנות כלל הניתן - הנקו' היא שניתן לחבר אותם חזרה ע"י מבנה פשוט. וזה מאפשר חלוקת העבודה בין צוותים שונים. מתחילים ביחידה הגדולה ומפרקים אותה.
2. **הרכבה Composability -** עיצוב של יצירת רכיבים שניתן לשלבם ליחידה מורכבת יותר. בניית הרכיבים בצורה מסוימת המאפשרת שימוש במקומות שונים לגמרי. מתחילים מיחידות קטנות וממשיכים ליחידה גדולה כאשר כל רכיב אוטונומי.
   ברגע שבונים יחידות שניתן להשתמש בהם בהקשרים שונים - מאפשר את רעיון מחזור הקוד.
   דוגמא קלאסית: חבילות התוכנה collection frame work לדוג'. חבילה המורכבת מרכיבים מסוימים שניתן להשתמש בהרבה אופנים.

פריקות והרכבה הפוכות בסדר חשיבתם אך לרוב ניתן להשתמש בשני העקרונות ביחד וביחד ליצור פתרון מיוחד.

3. **מובנות** - דיזיין מודולרי משמעותו שקורא אנושי יכול להבין כל מודול בנפרד ובלי צורך להבין לעומק את שאר המודולים האחרים.
   כלל אצבע העוזר לדעת אם אנו עומדים בכך הוא האם אנו יכולים לתאר את המערכת בכמה מילים.
   מובנות אינה זהה לקריאות - קריאות היא ברמת הקוד ומובנות היא ברמת העיצוב.
4. **רציפות -** במידה ויש שינוי בדרישות - המודול הזה ישפיע על כמה שפחות מודולים אחרים. דורשים שכל שינוי במודל מסוים לא ישפיע הרבה על סביבתו (זהה למושג של רציפות - שינוי קטן משפיע בסביבה קטנה).

open close principle, Single choice principle

open close principle - עקרון תכנות לפיו רכיבי תוכנה צריכים להיות סגורים לשינויים אך פתוחים להרחבה.
תוכנה שבה עדכון של התכונה כמו הוספת אפשרויות חדשות וכדו' גורם לשינוי בחלקים אחרים בתוכנה, הינה בעלת דיזיין שלילי. שיטת העיצוב הנכונה אינה על ידי שינוי הקוד הקיים כי אם על ידי <u>הוספת</u> קטע קוד חדש כך שמודולים העובדים לפי עיקרון זה הם :

- פתוחים להרחבה - ניתן להרחיב את פונק' התוכנה.
- סגורים לשינויים - קוד קיים שכבר עובד לאחר בדיקות באגים ומופעל אצל לקוחות לא נרצה לגעת בו – שהרי שינויים דורשים דיבוג מחודש וכדו'. לכן הוספת קוד חדש מאפשרת בדיקה פרטנית של הקוד הנוסף בלבד - דבר הקל מכל הבחינות.

Single choice principle – עקרון תכנות לפיו במקרה שבו אנו נדרשים לבחירה מרובת אפשרויות - אנו נרצה שהיא תהיה מוכרת רק במקום אחד. כך שגם במקרה שבו נדרש להוסיף אפשרות נוספת - השינוי שנעשה ייעשה רק בקטע קוד אחד. ורק שם נצטרך להוסיף את האפשרות השנייה.

Factory design pattern

Factory - אובייקט המשמש לבניית אובייקטים אחרים.

מאפיינים:

- יכול לקבל ערכים לטובת יצירת הערכים.
- סוג הדיזיין זה הוא כללי - כלומר ישנם כמה סוגי דיזיין פרטיים הנובעים מהדיזיין הזה.
- עיקרון זה מתקשר באופן ישיר לשני העקרונות הקודמים של open-close ו single choice .

Singleton – מטפל במקרה שאנו רוצים ליצור אובייקט אחד ויחיד. לדוג' task manager.
עיצוב זה קשור לאובייקט היצירה.
האינסטנס שייווצר:

- אחד ויחיד
- קל לגישה.

תנאים:

1. מאחסן אינסטנס יחיד בתור private static - עונה על הדרישה : אובייקט אחד ויחיד.
2. הקונסטרקטור מסוג private . עונה על הדרישה: לא ניתן ליצור נוספים.
3. מתודה סטטית יצירת אובייקט - הנקראת instance() - גישה נוחה.

דוגמא:

```
public class Singleton {
    private static  Singleton single =
                                    new Singleton();

    private Singleton () { ... }

    public static  Singleton instance() {
            return single;
    }
}
```

השלכות:

- לא ניתן לרשת מהמחלקה הזו שהרי כאשר מבצעים ירושה בין מחלקות, במחלקה היורשת יש איזשהו "חלק" מהמחלקה המורישה מה שאינו מתאפשר במחלקת ה singleton שהרי לא ניתן לקרוא לקונסטרטור ה-private שלה.
- ניתן להציע פתרון של מחלקה בה כל המתודות הן סטטיות. אך היתרון הוא שבסינגלטון ניתן להשתמש בירושה ובפולימורפיזם (כמו מימוש אינטרפייס). בנוסף הלקוח יכול אפילו לא לדעת שהמחלקה היא singleton.

Strategy

עיצוב שבו למערכת מסוימת יש משפחה של אלגוריתמים או התנהגויות המאפיינים את פעולת המערכת והעיצוב מאפשר להחליף בין האלגוריתמים השונים - גם בזמן ריצה.

- דיזיין מסוג התנהגותי.

לדוג', נרצה תכנית שבה נוצר מיון ואנו נעדיף למיין לפי אלגוריתמים שונים כל פעם.
פתרון: הגדרת מחלקה כללית או אינטרפייס עם api כללי. ונממש את ה- api בצורה ספציפית לכל אחת מהאופציות.
והבחירה בין האלגוריתמים השונים יימצא במחלקת ה factory.

- בדומה לפקטורי, גם סטרטגי שומר על עקרונות על open closed ו single choice.
- יתרון בstrategy שומר על מודלוריות ויוצר החבאת מידע (שהרי ללקוח לא בהכרח חשוב מה האלגוריתם שבו אני משתמש), מחזור קוד ומאפשר שינוי אלגוריתם בזמן ריצה.

הקשר בין strategy, factory, singleton – באופן כללי אלגוריתם ההתנהגות יהיה מסוג singleton שהרי לרוב לא יהיה לנו צורך ביותר ממופע אחד של האובייקט.

Functional interface

interface בעל מתודה אבסטרקטית אחת (אך יכול להכיל מתודות דפולטיביות או סטטיות הממומשות במלואן). ניתן לשים את האנוטציה @FunctionalInterface וכך הקומפיילר יוודא שזה אכן functional interface.
ניתן ליצור אינטרפייס כזה ע"י local class או באמצעות anonymous (לדוג' בתוך מתודה היוצרת אובייקט מסוג האינטרפייס הזה – כאשר את הlocal – נממש מחלקה פנימית הממממשת את האינטרפייס ונחזיר אינסטנס שלה new localClassImplementsFI או באמצעות מחלקה אנונימית שנחזיר 'ישר' את המחלקה הממממשת את ה אינטרפייס).

Local class	Anonymous class

`public static BinaryOperator<Integer> getAddition(){`
`  class Addition implements BinaryOperator<Integer>{`
`  @Override`
`  public Integer apply(Integer t, Integer u){`
`  return t+u;`
`  }`
`  }`
`  return new Addition();`
`}`
	`public static BinaryOperator<Integer> getAddition(){`
	`  return new BinaryOperator<Integer>() {`
	`  @Override`
	`  public Integer apply(Integer t, Integer u){`
	`  return t+u;`
	`  };`
	`}`

Lambda expressions

ביטוי המשמש בתור קיצור למתודה אנונימית. המבנה: (parameter list) -> {statements.. return something;}.
למבדה שימושית מאוד במקרה של functional interface. (במקרים פשוטים בעיקר).
ענייני סינטקס:

- אין צורך לכתוב את הטייפ של הפרמטרים והם יקבעו בהתאם להקשר.
- ביטוי למבדה המכיל ביטוי אחד לא חייב להכיל את משפט ה return.
- ביטוי המכיל פרמטר אחד אינו צריך להכיל סוגריים.
- ביטוי שאינו מקבל פרמטרים נכתב בתור סוגריים ריקים ().

הרצאה 9

intro to streams

התוכנה שלנו היא הגורם המתווך והעומד באמצע בין זרימת מידע. בין מקור קלט לבין מקור שאליו אנו פולטים. וה-stream זה הרעיון העומד בתווך הזה.

כוון שיש הרבה מקורות מידע מסוגים שונים, התקשור ביניהם הוא מלאכה מורכבת וצריך לחשוב כיצד ליישם זאת בצורה יעילה. הצורה הפשוטה היא ליצור Api נפרד לכל פעולת תקשורת בין שני מקורות, אך הדרך הטובה ביותר היא לשאול את המשותף לדרכים האלו ולאחד אותם ב api יחיד.

מה שעומד בראש הפעולות האלו הוא קבלת מידע וקליטתו. קריאה וכתיבה.

Java stream library - מאפשר עבודה אבסטרקטית של זרימת המידע. והיופי במערכת הזו היא שהיא מחביאה את המקורות והצינורות - המתכנת לא צריך לדעת מהו סוג המידע אלא את עצם זה שהוא יכול לקבל ולהעביר - אנלוגי לברז : אדם הפותח בארז לא משנה לו מאיפה הגיעו המים: ים, הטפלה וכו'.

מתודות בהן נשתמש :

- Create stream
- Write data to stream
- Read data from the stream
- Delete stream

פסאודו קוד של עבודה עם stream:

1. Open a stream to - file etc
2. While(more data)
   a. Read/write
3. Close the stream

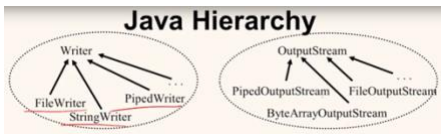
מהו מידע ?

ברמה פשוטה ניתן להפריד סוגי מידע לשניים: טקסטואלי ובינארי.

- טקסטואלי - מידע שמכיל רצפי אותיות - קריא בידי אדם. דוגמאות: קבצי טקסט, קבצי אינטרנט, קבצי java .
- בינארי - רצפי 0 ו 1 כמו קבצי תמונות מוסיקה: jpeg, mp3, class, zip.

Data encoding - העברת מידע מטקסטואלי לבינארי או בין שתי שפות. וכמו בתקשורת בסיסית יש לקבוע כללי שפה שבה אנו מחליטים כיצד מתבצעת ההמרה בין שתי השפות. לדוג' בקבצי html כל קובץ מתחיל ב <html> .

- בתקשורת המתבצעת בstreams קיימת אחריות על שני צדדי המתקשרים לקבוע ולעמוד בכללי הקידוד.



Java.io - חבילה שבה מוגדרים ארבע מחלקות אבסטרקטיות המחולקות ל 2 סוגים:

1. reader ו writer המיועדות לעבוד עם טקסט.
2. input stream ו outputstream. המיועדות לעבוד עם מידע בינארי.

Reader נותן api ומימוש חלקי עבור סטרים לקריאת תווים ו- writer עבור כתיבה תווים.

מחלקות לעבודה עם streams:

Stream Overview

I/O Type	Streams
Memory	CharArrayReader/Writer ByteArrayInput/OutputStream
Files	FileReader/Writer, FileInput/OutputStream
Buffering	BufferedReader/Writer, BufferedInput/OutputStream
Data Conversion	DataInput/OutputStream
Object Serialization	ObjectInput/OutputStream
Filtering	FilterReader/Writer, FilterInput/OutputStream
Converting between bytes and characters	InputStream/OutputReader

טיפול בשגיאות:

```
try {OutputStream output = new FileOutputStream(args[1]);
    InputStream input = new FileInputStream(args[0]);} {
    int result;
    while ((result = input.read()) != -1) {
        output.write(result);
    }
}catch (IOException ioe) {
    System.err.println("Couldn't copy file");
}// No need to close streams! (AutoCloseable interface rocks!)
```

באופן כללי נשים את קטע קריאת הקובץ ב catch-try כוון שכל שלב בעיסוק בקבצים יכול לגרום להצפת שגיאה.
במקרה שבו עלתה שגיאה לפני שהקוד הגיע לשורה של סגירת הקובץ - לשם כך יש פתרון נחמד בגרסת java7 ומעלה. הפיצ'ר דואג לסגור את הקובץ ברגע שעברנו מה try ל catch:
עניין נוסף הוא עניין של קריאות, שהקורא מבין שרק בקטע הזה קיים עיסוק בקריאה ובגישה לקבצים הנקראים ולאחר מכן הם נסגרים ולא נעשה בהם שימוש.
<u>זו הדרך המומלצת לעסוק בקבצים.</u>

decorator

Decorator הוא design pattern שיש לו קשר ל streams - אך לא רק.
הבעיה: נניח ואנו רוצים תוכנה שכותבת מידע ל stream אך נעדיף לכתוב כמה שפחות מכמה סיבות כמו להקל על חבילת הגלישה וכדו'.
הפתרון לכך הוא קיבוץ המידע. - במידע יש הרבה פעמים הרבה יתירות וישנן המון שיטות של קיבוץ מידע.
הבעיה בפתרון : אנו מעוניינים למסור מידע מקובץ להרבה גורמים ואם נעבוד בצורה נאיבית היינו לוקחים כל מחלקה קונקרטית של io ומרחיבים אותה, אך באופן הזה קיימת חזרה רבה על הקוד ונוצר קוד הקשה לתחזוק.
בעיה 3: נניח ואנו רוצים לקחת קובץ מאוד גדול ואנו רוצים לקרוא אותו byte byte וכן להפך בכתיבה. הבעיה היא שפעולת כתיבה וקריאה הן פעולות יקרות. אך כאשר אנו כבר קוראים אין משמעות רבה ל**כמות** שאנו קוראים.
לכן הפתרון לבעיה הזו הוא זהה לפתרון של פח ביתי, צבירת אשפה בפח הקרוב - פעולה זולה וברגע שהפח מתמלא נלך ונזרוק לפח השכונתי - פעולה יקרה. וזהו בדיוק רעיון ה-buffer .
כאשר אנו רוצים לכתוב הרבה מידע ל stream אחר. לא נכתוב אותו בייט-בייט אלא נכתוב את כל המידע אל ה buffer, וממנו נעביר את כל המידע בפעם אחת.
החיסרון בכך הוא עיכוב בין הזמן שהמידע נכתב לבין הזמן שהוא מועבר אל המקום שאנו רוצים (לדוג' מזון מקולקל לא רצוי להשאיר עד שמתמלא) אך ברובם המוחלט של המקרים זה יעיל יותר.
בעיה קודמת: אנו מעוניינים שקריאה וכתיבה יעילה יהיו מתאים עבור הרבה סוגי מידע ואפשרויות.
נשים לב שאלו שני חסרונות התלויים זה בזה : כלומר קיבוץ המידע והעברתו בפעם אחת מחזקים זה את זה.
שוב באמור, מימוש לכל מחלקה יוצר מספר עצום של מחלקות שיש ליצור וזאת בנוסף למקרים שבהם אנו לא נרצה להשתמש באופציה הזו.
הגדרת הבעיה בצורת design - להרחיב streams עם יכולות נוספות.
נושאים: הרבה הרחבות, הרבה סוגים של input and output stream ויצירת כפל קוד משמעותי.

פתרון ה decorator

אנלוגיה ל-decorator היא שקע ותקע: השקע הוא מקור המידע, הכבל המאריך הוא היכולת הנוספת שאנו רוצים (כמו כיווץ או כתיבה באמצעות buffer).
פורמלית: אנו מעוניינים לקחת מחלקה B ואנו רוצים לבנות מחלקה A שהיא תהיה מחלקת ה buffer כאשר:

- A מרחיבה את המחלקה b - חולקות את אותו ה api .
- האצלת בקשות ל b . (כמו הכבל המאריך שלא מייצר חשמל בעצמו אלא מקבל אותו מ b).
- מאפשר "שרשור".

הdecorator קולט אובייקט בקונסטרקטור ובכך משתמש בו.
המחלקות המקשטות אינן מקורות מידע משל עצמם הן רק מעבירות בקשות ממחלקות שיש להן את המידע הקונקרטי.
המחלקות המחזיקות את המידע עצמו הן אינן decorators.

- הן מייצגות מקור מידע ולא פונקציונאליות
- וגם לא ברמה הפרקטית, הן לא עושות האצלה לאובייקט מסוג stream אחר.

סיכום של הכלה, האצלה וקישוט:

1. A composes B – A מחזיק instance של b בתור data member או local variable.
2. A composes B – A delegates B ובנוסף מעביר 'בקשות' ל -B.
3. A delegates B and extends B – A decorates B ובנוסף מוסיף תכונות נוספות.

לדוג': Scanner מחלקה שמשמשת לקריאת מידע טקטסואלי והיא מקבלת בקונסטרקטור רכיב של אינפוט סטרים ומעבירה בקשות למחלקה הזו. מחלקה זו שימושית מאוד עבור יצירת אנליזה לטקסט כמו קריאת שדות של טקסט באמצעות מפרידים וכדו'.

- סקאנר משתמש בבאפר קטן.
- סקאנר משתמש בdelegation - הוא מעביר בקשות לאינפוט סטרים שאיתו הוא עובד אך מצד שני הוא לא דקוריישן כוון שהוא לא יורש מאינפוט סטרים ולכן אי אפשר להרכיב עליו תכונות נוספות.

Enum

Enum הוא type בג'אווה (כמו מחלקה ואינטרפייס). האינסטנסים של אובייקט enum מוגדרים עוד בזמן כתיבת הקוד ובלתי ניתנים לשינוי ולכן ה-enum מאפשר זיהוי של שגיאות בבחירת סוג שגוי עוד בזמן הקומפילציה.
אינטסטנס של enum בנויים בדומה למחלקה רגילה כאשר ה-instence-ים שלו מוגדרים בתחילת המחלקה. וכל enum ספציפי משמש כאובייקט של מחלקה מבחינת קריאה לקונסטרקטור, מתודות וכדו'.
הקונסטרקטור של הenum מוגדר בדיפולט כprivate or package.
המתודה ()EnumName.values מאפשרת לרוץ על ערכי ה enums.
במקרה של בחירה מרובה של enums על מנת לשמור על ה-single choice principle ניצור מתודה אבסטרקטית ל ENUM וניצור מימוש למתודה זו בתוך כל enum (ע"י סוגריים מסולסלים ליד שם ה- enum) או ע"י יצירת functional interface כאשר כל enum מכיל בתוכו אובייקט של ה-FI וברשימת ה enums נוכל לקרוא לכל enum באמצעות הקונסטרקטור (סוגריים עגולים).
על מנת לקבל enum ספציפי נשתמש במתודה : valueOf לפי הסיטנקס הבא: EnumName.valueOf(string of enum instance)

הרצאה 10

Generics

Generic הם abstract non primitive type - הפשטת טייפים לא פרימיטיביים.
המטרה המרכזית של generic הוא להפוך אותו להיות type safety ובנוסף הוא מוסיף קריאות.
עקרון "מניעה אל מול טיפול" בתכנות - ככל שמזהים טעות בשלב מוקדם יותר נשלם מחיר זול יותר ועקרון זה בא לידי ביטוי במנגנון ה-generic המאפשר לזהות שגיאות בשלב מוקדם יותר - שגיאות מסוג type errors.
בגדול ישנם שלושה סוגי טיפוסים - פרימיטיביים, מחלקות ומערכים (היכולים להיות גנריים).
Type safe הוא מצב שבו התוכנית יודעת להגן על עצמה משגיאות type error בזמן **קומפילציה.**

יצירת מחלקה גנרית
פרמטר גנרי מוגדר בתוך הגדרת המחלקה (עם הסוגריים המשולשים).
בתוך המחלקה T משמש בתור טייפ קונקרטי - **המוגדר בזמן ה-instantiation**.
generic הם invariant - אם יש מחלקה או אינטרפייס בעלי שני טיפוסים שונים שעל אף שאחד הוא sub-type של השני, המחלקות הגנריות אינן שומרות על היחס הזה, כלומר:
LinkedList<String> does not extend LinkedList<Object>. והשורה הבאה תביא לשגיאה קומפילציה:

```
LinkedList<Object> myObjList = new LinkedList<String>();
```

אין משמעות הדבר שלא ניתן **להכניס מחרוזות לתוך רשימה מקושרת של אובייקט** אלא שהרפרנס של המחלקה לא יכול להחזיק אובייקט מטיפוס שלו שונה.

כמה הגבלות:

1. לא ניתן ליצור אינסטנס עם טייפ פרמיטיבי.
```
List<int> l = new ArrayList<int>();
```
2. לא ניתן ליצור אינסטנס עם הטייפ הגנרי:
```
E e = new E();
```
3. לא ניתן ליצור מערך של ביטויים גנרים – כפי שנפרט בהמשך.
```
List<Integer>[] l = new LinkedList<>[2];
```

Bounded Type parameter – כאשר אנו רוצים ליצור משתנה עם 'תקרת' משתנה מסוים נוכל להשתמש בסיטנקס הבא: E extends Class. רעיון זה דומה ל wildcard הבא:

Wildcards

על מנת לפתור את המוגבלות שנובעת מהחלק הקודם נציע את הדרך הבאה:
במקרים שבהם אנו לא יודעים (או שלא משנה לנו) מה הסוג הספציפי של אובייקט מסוים נוכל להשתמש בסימן השמור בג'אווה " ? " המאפשר להכניס כל סוג של אובייקט. אך במקרה זה אנחנו לא יכולים להניח שום דבר על הרשימה הזו. לכן ניתן לעשות שני דברים בלבד:

1. שניתן להניח הוא שהאובייקט הקיים יורש מ-object וממילא להשתמש בפעולות השייכות ל-object
2. ניתן להוסיף לרשימה הזו רק null .

נעדיף להשתמש במקרה הזה כאשר:

- אנו רוצים ליצור פעולות מסוימות אך לא משנה לנו איזו רשימה נכנסת - כמו מחיקה או הדפסה. או מתודה הבודקת האם שתי רשימות מאותו גודל.
- במקרה בו נרצה להחזיר פרמטר לא ידוע בזמן כתיבת הקוד כי אם רק בזמן ריצה (לדוג' switch).
- Type-safe (לעומת list<object>).

תכונות:

- לא ניתן להוסיף לרשימה כוון השאובייקט יכול להיות כל דבר (כל מחלקה וכו').
- ניתן להחזיר אובג'קט כוון שכל אובייקט יורש ממנו ולכן ניתן להחזיר אותו.

- ניתן להשתמש ב"?" רק ברפרנס אך אתחול של משתנה עם "?" חייב להיות משתנה קונקרטי.
  כלומר השורה הבאה אינה תקינה : List<?> list = new LinkedList<?>();

? Extends MyClass

ביטוי שימושי נוסף המאפשר ליצור גנריות עם 'תקרה' מסוימת שכל סוגי הרשימות שיוכנסו יורשות ממנה. לדוג':
LinkedList<? Extends Animal> myList = new LinkedList<Dog>(); // legal

יתרונות:

- מאפשר לבצע פעולות השייכות למחלקת הראשית (ה- MyClass בכותרת) או לאינטרפייס.

תכונות:

- יכול להיות מאותחל ע"י המחלקה הראשית או המחלקות היורשות ממנה.
- ניתן להוסיף רק null .
- ניתן לשלוף את המחלקה ומי שמעליה.

Erasure

הדרך שבה ממשים ג'נריקס ב ג'אווה - תהליך שבו הג'נריקס מומר ומחליף אותו בפרמטר המתאים.
מה הצורך בשימוש הזה אם בכל מקרה כל הדברים נמחקים ?
תהליך ה erasure מתרחש רק אחרי שהקומפיילר וידא שאין שגיאות type error ולכן אחרי שהדברים כבר "בטוחים" לא כל כך משנה לנו שהקומפיילר ישנה את זה - בקיצור, הרעיון של ג'נריקס מועיל ת'כלס בשביל המתכנת ובשביל מי שקורא את הקוד שלו ולכן לא משנה מה קורה אחר כך.

Array[] הוא covariant. כלומר עבור מערך המכיל מחלקה a, כאשר a היא sub-type של A, אזי a[] הוא sub-type של A[]. כלומר השורה הבאה תקינה: A[] Aarr = new a[].
אך זה יכול לגרום לשגיאת זמן ריצה: כמו דוגמת הקוד הבאה:

```
String[ ] strArray = new String[10];
Object[ ] objArray = strArray; // Allowed due to covariance: String extends Object
objArray[0] = new Integer(5);
```

אך כוון שג'נריקס נועד למנוע type error - ג'אווה אסרו מערכים של ביטויים גנרים כלומר הביטוי הבא לא קיים : LinkedList<String>[]...

השפעת ג'נריקס על הקוד:

- אינו משפיע על הקוד מעבר לצורת הכתיבה.
- קוד שאינו גנרי יכול לעבוד עם ספריות גנריות.

יש לציין, קוד גנרי אינו קוד כללי - ואינו מבצע דבר נוסף שלא יכולנו לעשות לפני כן. הדבר היחידי הוא שהוא עוזר לנו למנוע type errors ולהפוך את הקוד לקריא יותר.

Generic Methods

מתודה בעלת טייפ מסוים.
public <T> void doSome(T arg)
כוון שהטייפ הגנרי מוגדר בזמן ה-instantiation, מתודה <u>סטטית</u> אינה יכולה להשתמש בטייפ הגנרי של המחלקה ולכן צריך להגדיר טייפ למתודה עצמה.

אך מתודה שאינה סטטית, נוסף על האפשרות להגדיר אותה כג'נרית, היא יכולה להשתמש בטייפ של המחלקה.

הרצאה 12

serialization

serialization הינו תהליך של העתקת אובייקט ו'העברתו' למקום אחר בדומה להעברת קבצים במחשב וכו' - ניתן לעשות זאת גם על אובייקטים בג'אווה.
ה-framework שבו עובדים על מנת לבצע את התהליך הוא ה-streams. נסתכל על אובייקט כסוג של מידע ונעביר אותו מהתוכנה שלנו אל תוכנה אחרת.
Deserialization - התהליך ההפוך של קבלת אובייקט ולפרק אותו למשתנים שלו.
באופן דיפולטיבי מחלקה לא מוגדרת בתור כזו שיכולה להישמר, ועל מנת לאפשר זאת עליה לממש את ה- marker interface ה-serializable. וכוון שהתהליך הוא רקורסיבי (שהרי ישנה העתקה של תתי השדות וחוזר חלילה) אז כל שדה צריך להיות גם כן serializable.
קטע קוד לדוגמא:

```
try (OutputStream out = new FileOutputStream( "save.ser" );
        ObjectOutputStream oos = new ObjectOutputStream(out);) {
                oos.writeObject( new Date() );
} catch (IOException e) {... }
...
try ( InputStream in = new FileInputStream( "save.ser" );
        ObjectInputStream ois = new ObjectInputStream( in );) {
                Date d = (Date) ois.readObject();
} catch (IOException e) {... }
```

These are *decorating* classes

Down-casting required

intricacies of serialization

כאמור, תהליך הserialization מבצע העתקה של אובייקט, אך מה קורה במקרה שכבר שמרנו שדה מסוים כבר בעבר ? (בדוג' חבר של חבר עד שחוזרים לאותו אובייקט) - שמירת אותו אובייקט בזכרון אינה משנה את האובייקט גם במקרה שבו נוצר שינוי באובייקט בין השמירות וזאת משום שבשמירה הבאה נשמר הרפרנס לאובייקט והרפרנס הזה לא משתנה כשהאובייקט נשמר שוב.

- על מנת ליצור שינוי באובייקט עצמו בשמירה נוספת, הפתרונות פשוטים הן לסגור ולפתוח שוב או לעשות reset - אך אלו פתרונות לא יעילים במיוחד.

Transeint - מילה שמורה האומרת ל-stream שאלו שדות שאיננו מעוניינים לשמור. בתהליך ההעברה, מדלגים על השדות האלו ובתהליך הקריאה הם מאותחלים לערך הדיפולטיבי שלהם.

- שימושי במקרה שאין צורך אמתי בלשמור שדות מסוימים.
- שדות סטטיים - אין משמעות לשמור שדה סטטי שהרי זהו שדה השייך למחלקה ולכן אין טעם לשמור אותו באובייקט מסוים.

Primitives לא יכולים להיות serialized אך ניתן להשתמש במתודות של האינטרפייס DataOutput, הממומש ע"י ObjectOutputStream, המאפשרות כתיבה של primitives - הכוונה כמובן להמרה אקטיבית של ערך פרימיטיבי ולא ל-data members פרימיטיביים של אובייקט שאכן עוברים.
כאמור, שינויים באובייקט אינם נשמרים בהעברה מחודשת אך ישנם מקרים בהם השינוי הוא קריטי ודורש התייחסות (שינוי קריטי משמעותו שהמחלקה השתנה וכעת זו מחלקה שונה. אלו שינויים כמו שינוי טייפים של data members וגדו') עבור מקרים כאלה קיים משתנה סטטי השומר את הגרסא האחרונה של האובייקט לפני שינוי קריטי כלשהו, משתנה זה נקרא SerialVersionUID. זהו השדה הסטטי היחידי שנשמר.
כך שכאשר serialVersionUID השתנה משמעותו שהמחלקה השתנתה וכאשר הוא לא השתנה, מתייחסים למחלקה שהשתנתה בתור המחלקה המקורית.
במקרה שבו serialVersionUID השתנה:

- לא ניתן לעשות deserialization ל-object אל תוך המחלקה החדשה.
- כאשר מנסים לבצע deserialization ל-object עם גרסא שונה נזרקת שגיאת invalidClassException

Java מגדירה את המשתנה הזה באופן דיפולטיבי אך מומלץ להגדיר באופן מודע כוון שבכך ניתן להחליט איזה שינויים נחשבים 'קריטיים' ואלו לא לעומת java שמתייחסת לכל שינוי כקריטי. ומומלץ להגדיר אותו בתור private כוון שמשתנה זה אינו שימושי עבור המחלקות היורשות.

Cloning

Shallow copy יצירת אובייקט חדש שכל המשתנים שלו מצביעים לאותם שדות לעומת **deep copy** שהרפרנס מצביעים לאובייקטים חדשים.
על מנת שמחלקה תועתק היא צריכה 'להרשות' זאת ע"י מימוש ה-marker interface : cloneable ולדרוס את מתודת ה clone (השייכת ל Object)
מתודת ה- ()clone. מבצעת שני דברים:

1. בודקת האם האובייקט מממש את הcloneable – במקרה ומנסים לבצע פעולת ()clone על אובייקט שאינו מממש את האינטרפייס הזה תזרק שגיאת CloneNotSupportedException.
2. באופן דיפולטיבי מתודת clone מבצעת shallow copy.

נקודות נוספות:

- מערכים הם גם cloneable והם מבצעים shallow copy.
- על מנת לבצע deepcopy יש לדרוס את מתודת ה clone : בתחילה לבצע shallow copy באמצעות A a = (A) super.clone() ולאחר מכן לבצע ()clone על כל משתנה ספציפי שאנו רוצים שהוא יועתק בצורה עמוקה ולבסוף להחזיר את a.
  נשים לב כי מתודת ה clone מופעלת על אובייקט קיים ולכן זה ה ()super.clone הקיים באופן דיפולטיבי ב-Object, בתוכו יוצרת אובייקט חדש, ומחזירה אותו. החתימה : ()public Object clone.
- אלטרנטיבה טובה יותר לclone - copyConstructor. מקבלים בקונסטרקטור אובייקט מסויים ועליו מבצעים את השכפול, בין אם ע"י השמה פשוטה ובין אם ע" clone

Reflections

מנגנון המאפשר לתוכנה 'לשאול שאלות על עצמה'. לדוג' להדפיס את השדות של התוכנה עצמה.

1. זהו אחד הכלים החזקים וה"מסוכנים" (?) בקורס.

לכל reference type ג'אווה יוצרת immutable instance של java.lang.Class.
את המחלקה ניתן לקבל באמצעות הקוד:

```
Class class = Class.forName("ClassName");
```

אובייקטים שניתן לקבל:

Constructor list

1. פקודה: ()Constructor[] ctorList = cls.getDeclaredConstructors.
2. יצירת אובייקט חדש:
   Object retObj = ctorList[i].newInstance(arglist).

Methods

פקודה : ()Method[] methList = cls.getDeclaredMethods
פקודה זו מחזירה גם את המתודות הפרטיות.
ניתן לקבל גם נתונים כמו מי המחלקה שהגדירה את המתודה הזו וכן רשימת הפרמטרים שהמתודה מקבלת ע"י ()getDeclaredClass ו ()getParameterTypes בהתאמה.
ניתן להפעיל את המתודה ע"י : methList[i].invoke(obj, arglist). כאשר obj הוא instance של המחלקה. ומתודה זו מחזירה את ערך ההחזרה של המתודה ב- up casting ל object. ו- null במקרה של void.

Fields

הפקודה: ()Filed[] fieldList = cls.getDeclaredFields.
ניתן לאתחל את השדות האלו או לקבלם ע"י:
fieldList[i].**set**(Object classInstance, Object data) ו- fieldList[i].**get**(Object classInstance).
בנוסף, מאפשר לקבל מידע על הmodifiers
באופן דיפולטיבי אינו מאפשר גישה לשדות הפרטיים אך ניתן לעשות זאת באמצעות : Field.setAccessible(true).

מה הצורך ב Reflection?

גמישות והרחבה - מאפשר הוספת אובייקטים חדשים כאשר המחלקה המקורית אינה מודעת אליהם בזמן קומפילציה.
Serializable משתמש ב reflection.
Reflection סותר את עיקרון ה-encapsulation ויכול ליצור תופעות לוואי שגויות וכן זמן הריצה יורד משמעותית.

Syntax

Trenary condition:

Result = condition ? do if true : do if false