**OOP - פרק 7**

**7.1 מבני נתונים שימושיים:**

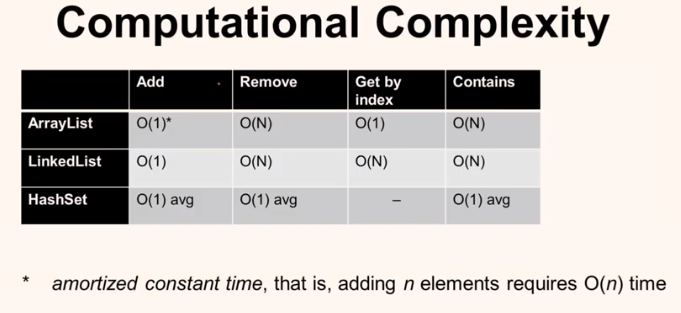
* **היכרות עם SETת LISTת MAP:**
  + הממשק הראשון נקרא LIST:
    - מדבר על רשימה שבה מוגדר סדר מסוים (כלומר המיקום של האיברים ברשימה חשוב)
    - נותן לנו גישה ע"פ אינדקס – לגשת עם get ולשנות עם set
    - יכול להכיל אותו איבר כמה פעמים.
  + SET:
    - ממדל את הקונספט של קבוצה מתמטית
    - לאיברים בקבוצה אין סדר וכל איבר יכול להופיע פעם אחת בדיוק
  + MAP:
    - מאפשר להגדיר מיפוי בין מפתחות לערכים
    - המפתחות חייבים להיות ייחודים – יופיעו פעם אחת בלבד
      * הערכים כן יכולים להופיע כמה פעמים
* **The Collections Framework:**
  + LIST מיוצגת בג'אווה ע"י הממשק List:
    - הוספת איבר – add
    - הסרת איבר – remove
    - גודל - size
    - בדיקת הופעת איבר ברשימה – contains
    - get, set
  + קבוצה – Set – מייצגת קבוצת איברים בלי סדר ובלי כפילויות מיוצגת ע"י הממשק Set:
    - הוספת איבר – add
    - הסרת איבר – remove
    - גודל - size
    - בדיקת הופעת איבר ברשימה – contains
  + ניתן לראות שיש דברים משותפים ביניהם.
  + Collection מייצג אוסף אין בהכרח סדר לאיברים שלו אבל יכול להיות, אין בהכרח כפילויות אבל יכול
    - כלומר אם יש לנו אוסף אנו יודעים שאפשר להוסיף ולהסיר איברים, לבדוק אם איבר קיים ולקבל את הגודל שלו.
    - הממשקים LIST וSET יורשים ממנו ומקבלים משם את השיטות האלו
  + Map – מפה או מילון, מבנה נתונים שממפה בין מפתחות לנתונים
    - הוספת איבר – put(key, val)
    - הסרה לפי מפתח – remove(key)
    - בדיקת קיימות – containKey(key)
    - get(key)
    - גודל – size()
    - keyset – מחזיר אובייקט מסוג Set של המפתחות (כי אין כפילויות)
    - Values – מחזיר אובייקט מסוג collection
  + כל אלה רק ממשקים שמגדירים מה מבנה נתונים אמיתי יקיים.
  + נדבר על מימוש ArraySet, LinkedList עבור List, HashMap עבור Map ו-HashSet עבור Set
  + כל אלה הם חלק מה collection framework
    - Collection יורש מ-iterable שאומר שאפשר לעשות איטרציה for each
    - יש עוד מימושים לכל הממשקים
* **מימושים של List:**
  + הממשק LIST מאפשר כל מיני סוגים של רשימות שונות
  + המימוש הכי נפוץ בג'אווה לרשימה הוא ArrayList:
    - מממשת את הממשק LIST וזו רשימה במובן הקלאסי של רשימה.
    - כשאנו יוצרים מופע חדש, ג'אווה מקצה לנו מערך, כשאנו מתקרבים למלא את כל גודל המערך ג'אווה מוסיפה
    - מאפשרת גישה לפי אינדקס
  + LinkedList:
    - רשימה שלא שמורה במקום אחד בזיכרון אלא רשימה שמורכבת מאיברים שונים – כל איבר מכיר רק את מי שאחריו.
    - היתרון של רשימה מקושרת הוא במחיקה והוספת איברים – כאשר נמחק איבר אנו צריכים לעדכן רק את האיבר שמלפניו וזה שאחריו במקום להזיז את כל מי שאחריו.
    - פחות יעילה מבחינת גישה כי צריך לעבור על כל האיברים שלפניו.
* **מימושים של Set ו-Map:**
  + HashSet:
    - HASH זו בעצם פונקציה מתמטית שיודעת לקבל כל קלט שנרצה בג'אווה ובאיזשהו אלגוריתם פנימי להוציא לו ערך מספרי.
    - איך זה קשור לסט?
      * יש לנו טבלה, וכל איבר שנכנס לה יכנס לא לפי הסדר אלא לפי הערך המספרי שהוציאה פונקציית הHASH
      * הגישה והחיפוש לרשימה כזו מאוד פשוטים בגלל פונקציית הHASH
      * אם שני ערכים מקבלים את אותו ערך מהפונקציה נקבל התנגשות – יש לג'אווה מנגנון שמטפל בזה והיא תדע לא להכניס אותם לאותו תא.
  + HashMap:
    - המנגנון שלו מאוד דומה בעזרת פונקציית הHASH
    - יש לנו טבלה וערכי המפתחות קובעים את המיקום של הערך שלהם בטבלה.
    - כלומר את המפתח אנו נכניס לפונקציית הHASH, נקבל ערך מספרי ושם נשים את הערך המתאים למפתח.
* **העשרה:**
  + נתונים בזמן ריצה של תכנה יישמרו במבני נתונים כמו אלה שדיברנו עליהם. מה לגבי שמירה של נתונים מחוץ לזמן הריצה של תכנה, כאלו שנשמרים כקבצים על הדיסק? המבנים שמאכסנים אותם נקראים בסיסי נתונים. למשל קובץ עם רשימה של שמות יכול להיות מכונה בסיס נתונים, אם כי הוא יהיה מימוש מאוד בסיסי (pun intended). מבני נתונים סקאלביליים ומורכבים יותר, למשל כאלו של מוסדות או שירותי אינטרנט, הם תחום ידע בפני עצמו, שלא נכלל במסגרת הקורס.

**7.2 מנגנונים בתכנות מונחה עצמים – מנגנון ה-Generics:**

* בג'אווה מבני הנתונים, פרט למערכים, לא יכולים לשמור פרימיטיבים, רק עצמים! לכן בג'אווה ישנן מחלקות Wrapper (או מחלקות עוטפות) לפרימיטיבים: המחלקה Integer עוטפת את הפרימיטיב int, המחלקה Double עוטפת את הפרימיטיב double וכולי. עטיפה, בהקשר זה, משמעה פשוט שלמחלקה Integer יש שדה מטיפוס int. כל המחלקות הללו הן immutable, למשל new Integer(4) הוא עצם שעוטף עד סוף חייו את הערך 4 (ובשביל לעטוף מספר אחר יוצרים עצם אחר). מבני הנתונים של ג'אווה לא יכילו אם כן ישירות את הפרימיטיב int, אלא ישמרו מופעים של המחלקה Integer.
* **שימוש ב-Generics:**
  + חשוב להכיר את Generics כי הcollection framework עושה שימוש נרחב בו.
  + ה-Generics מדבר על הציר הבא – אנו עובדים עם אותה רשימה אבל מה שמשתנה זה התוכן של הרשימה (לא על רמה מופשטת יותר של מחלקות) – כלומר על מה מחזיקות המחלקות.
  + GENERIC מאפשר להגדיר פרמטר אחד או יותר לכל מחלקה או ממשק
  + כשאנו מסתכלים על הAPI של המחלקות והממשקים הם יכולים להשתמש בפרמטרים האלה
  + כלומר למעשה הגדרנו רשימה או מפה שמחזיקה פרמטר כללי
  + List<E> , Map<E, T>
  + כל פעם שאנו יוצרים מופע של מחלקה המממשת את הממשקים האלו אנחנו נחליף את E במחלקה אחרת, כך אנו יכולים לקבל רשימה של מחרוזות, רשימה של מספרים וכ'.
    - כלומר בכל פעם אנו למעשה יוצרים מחלקה חדשה
  + איך נשתמש? כשניצור מופע מחלקה נחליף את E בשם של המחלקה הקונקרטית.
  + כלומר אנו נצטרך לממש את המחלקות השונות עם המחלקות הקונקרטיות שנרצה ואז נוכל להשתמש בהן
  + אם ניצור מופע לרשימת מחרוזות, לא נוכל להכניס איברים אחרים.
* **תרגול שימוש ב-Generics:**
  + על כל רשימה שנרצה ליצור מופע נהיה חייבים להוסיף <> עם הטיפוס של האיברים שהרשימה תשמור – גם עבור הטיפוס מהממשק וגם עבור רשימת המערך הספציפית (הם חייבים להיות בהתאמה)
  + בדומה עבור סט ומפה
  + נשים לב שה-collections framework של ג'אווה חייב לקבל מחלקה, לא יכול לקבל פרימיטיביים ולשם כך יש לנו מחלקות של ג'אווה שעוטפות פרימיטיביים.
  + לרוץ על אוסף כלשהו – for (type x: collection) – עוברת על האיברים עצמם
  + אם נרצה ליצור סט סטטי של איברים (כלומר לא משתנה, immutable) נכתוב Set.of(vals…) ואז לא נוכל לשנות אותו.
  + אם נרצה מבנה נתונים שאנו כן יכולים לשנות נוכל למשל עבור מערכים – Arrays.asList(vals…) ונקבל רשימה מיוטבילית.
* **העשרה:**
  + יעילות השימוש במחלקות העוטפות. נניח שאנחנו רוצים מערך של פרימיטיבים. אם נשתמש במערך, נוכל ליצור מערך int[ ], אבל מנגד בשביל להשתמש ב-ArrayList הטיפוס שלו יצטרך להיות Integer. במקרה כזה המערך הפנימי בתוך ArrayList לא יהיה של ints אלא של מצביעים לעצמים מסוג Integer, שכל אחד מהם עוטף int יחיד. לכן במקרה של פרימיטיבים, מערך פשוט כן יעיל יותר מ-ArrayList. אבל זה רלוונטי רק לפריטיבים ורק במקרים בהם 1) הרבה מהקוד סב סביב פעולות על המערך, ו-2) כשיש בעיית ביצועים. כשהשימוש במערך ספורדי ו\או כשאין לנו צורך לייעל את הקוד, מבני הנתונים האלו יהיו שקולים אפקטיבית מבחינת ביצועים, בזמן ש-ArrayList מספק יותר נוחות.

**השיטה hashCode:**

* **הכנסה וחיפוש בטבלת גיבוב:**
  + כל עצם בג'אווה יורש ממחלקה OBJECT ולכן יורש את השיטות שלה, הוא יכול לדרוס ולהחליף במימוש משלו.
  + הכרנו את השיטות: toString שמחזירה ייצוג מחרוזתי של האובייקט ו-equals שמקבלת עצם אחר ומחזירה האם הוא שקול.
  + כעת נכיר את השיטה hashCode – מחזירה ערך מספרי שמייצג את העצם
    - מבני נתונים יכולים להשתמש בה כדי לקבל ערך מספרי שמייצג את העצם, כאשר תרצה להכניס או לחפש אותו במבנה הנתונים
    - מימוש ברירת המחדל של OBJECT עבור השיטה הוא להשתמש בכתובת בזיכרון
  + בעזרת שיטה זו מבנה נתונים כמו HashSet היא יכולה למפות את העצם שהלקוח ירצה להכניס לסט.
  + השימוש בשיטה מתקיים גם בהכנסת איבר לסט וגם בבדיקת contains
* **מימוש השיטה hashCode:**
  + אם יצרנו שני אובייקטים שקולים (מבחינת השיטה equals)
  + האם שני אובייקטים אלו ימופו לאותו תא?
  + חוק – יהיו שני עצמים שקולים לפי המימוש של equals אזי ה-hashCode שלהם צריך להיות זהה.
    - כלומר אם דרסנו את equals אנחנו צריכים לדרוס גם את hashCode, והמימוש החדש שלה צריך להתבסס על אותם שדות כמו equals.
  + אם להרבה עצמים שונים יש את אותו hashCode זה גורם להאטה בביצועים של טבלת הגיבוב.
  + ג'אווה פתרה את הבעיה של הגדרת hashCode ייחודי עבור צירוף של שדות - ישנה מחלקת utility בשם Objects עם השיטה hash שמקבלת כמה שדות שנרצה ועושה בדיוק את מה שצריך.
* **סיבוכיות במבני נתונים:**
  + List<E>:
    - ArrayList<E>:
      * Get, set – זמן קבוע
      * Contains, indexof, remove – O(n)
      * Add – זמן קבוע amortized (הוספת n איברים היא ב-O(n) אבל הוספת איבר אחד יכולה להיות יותר)
    - LinkedList<E>:
      * Get, set, contains, indexof, remove – O(n)
      * Add – זמן קבוע
  + Set<E>:
    - HashSet<E>:
      * הסרה, חיפוש והכלה לוקחות בממוצע O(1)
        + למרות במקרה הגרוע יכול להיות גדול מאוד
      * חשוב לזכור שאין הבטחה של סדר הדברים
  + HashMap<E> מבוסס על האשסט



* + פעולת ההוספה די יעילה בכל מבני הנתונים
  + גישה ע"פ אינדקס (רלוונטי רק לרשימות) – יש הבדל גדול בין שני סוגי הרשימות
    - לכן אם חשוב לגשת לפי אינדקס נעדיף את arrayList
  + חיפוש – עדיף להתמקד באובייקטים מסוג סט
* הערה - היחידה הזו עסקה בסיבוכיות של פעולות, אבל לא בעלויות הזמן שלהן. סיבוכיות היא מדד מתמטי שמתאר אך ורק את קצב הגידול האסימפטוטי של איזשהי פונקציה. מה שסיבוכיות לא מודדת זה את הערך של הפונקציה עבור מספר איברים נתון. במילים אחרות, יש קשר רופף בלבד בין הסיבוכיות של פעולה לכמה שניות בפועל היא תיקח עבור מבנה נתונים ספציפי. בעולם האמיתי פעולות עם סיבוכיות O(n) עשויות גם לקחת משמעותית פחות שניות מפעולה עם סיבוכיות O(1).
* **סיבוכיות מקום:**
  + על פניו, ב-ArrayList יש ברוב המקרים הקצאה של מקומות מיותרים במערך! אבל בפועל ההקצאה הזו מתגמדת ביחס לצריכת המקום של רשימה מקושרת.
  + אם תחשבו על כך, כל צומת ברשימה מקושרת דורש לא רק רפרנס לעצם, אלא גם מצביעים קדימה ואחורה ברשימה. די בהם בשביל לקבוע שעלות המקום של רשימה מקושרת תמיד גבוהה יותר, אפילו כשמספר האיברים המיותרים ב-ArrayList המקביל הוא במקסימום. בפועל, כל צומת צורך אפילו עוד יותר מקום משלושה רפרנסים בלבד. צריכת מקום גבוהה יותר מיתרגמת גם לתקרות זמן גבוהות יותר של ניהול זיכרון.
* **העשרה:**
  + ראינו שמבני נתונים משתמשים במנגנון ה-Generics של ג'אווה. List, למשל, מקבלת בסוגריים משולשים פרמטר-טיפוס.
  + אנחנו אומרים שטיפוסים עם פרמטר טיפוס הם טיפוסים גנרים (generic types).
  + אבל למה צריך את מנגנון ה-Generics? בואו נציע חלופה: אם כל מה שאנחנו רוצים זה List שמסוגל להכיל איברים מסוג כלשהו, היינו יכולים להיעזר בפולימורפיזם. במימוש כזה, היינו מגדירים מחלקה לא גנרית בשם List עם השיטה:

public void add(Object element)

* + בדומה ל-Generics, גם API כזה יאפשר לנו להגדיר רשימות של כל טיפוס שנרצה. אמנם, מסתבר שעד גרסה 5 של ג'אווה זה אכן היה המימוש היחיד למבני נתונים. בואו נשווה בין שתי החלופות: מחד - מחלקה לא גנרית שעובדת עם הטיפוס Object, ומאידך - מחלקה גנרית שממנה אפשר לצקת מחלקות קונקרטיות.
* **מחלקה גנרית לעומת מחלקה שעובדת עם Object:**
  + ברשימה מקושרת, אם נרצה להגיד לnode עם איזה איבר הוא עובד נשתמש בתכנות גנרי – נוסיף פרמטר T
    - בכל פעם שבמימוש רגיל השתמשנו ב-object נחליף אותו ל-T
  + בשימוש בפרמטר גנרי במקום ב-object אנחנו יכולים למנוע טעויות (אם חשבנו שיש ערך מטיפוס אחד אבל הערך הוא מטיפוס אחר)
  + כמו כן, לא נצטרך לעשות downcasting
  + פרמטרים גנריים:
    - כשאנו מגדירים מחלקה או ממשק חדשים נוכל להגדיר אותו להיות גנרי בעזרת <> עם פרמטר אחד או יותר.
    - כאשר אנו משתמשים ב-T בתוך המחלקה אנו למעשה משתמשים בטיפוס שהוגדר מראש – כלומר T יוגדר בשורת ההצהרה וכשנשתמש בו אנו למעשה מבקשים אובייקט מאותו סוג.

**7.3 שגיאות:**

* **סוגי שגיאות:**
  + נחלק שגיאות ל-4 סדרים:
    - שגיאות קומפילציה – שגיאות שניתקל בהן בזמן כתיבת הקוד (הפרת חוקי תחביר)
    - שגיאות שתוכנה מזהה – שגיאות שהקומפיילר לא מזהה אבל בזמן ריצה יש תוכנה שמזהה אותן (למשל מערכת הפעלה, JVM וכו')
    - באגים ידועים - הקומפיילר לא זיהה וגם תוכנה לא זיהתה, כלומר לא בעיות תחביר או בעיות של תוכנות – באג שלא הקריס את התוכנית אבל התוכנה רצה ועשתה משהו לא נכון.
    - באגים לא ידועים – באגים שלא הקומפיילר, לא התוכנה ולא אנחנו זיהינו.
  + אנחנו מעדיפים לתפוס שגיאות כמה שיותר גבוה בהיררכיה –
    - שגיאות קומפילציה – אנו יודעים מה השגיאה ואיפה, בדרך כלל אפשר לחפש את השגיאה. זמן הדיבוג קצר ובאופן יחסי הן קלות לפתרון.
    - שגיאות שתוכנה מזהה – ברוב המקרים התוכנה תדע להגיד מה הבעיה, לפעמים השגיאה יכולה להתגלות הרבה אחרי שורש הבעיה אבל לא תמיד, כלומר לפעמים נדע איפה הבעיה. זמן דיבוג בינוני.
    - באגים ידועים – אחנו יודעים שיש בעיה אבל אין לנו שום מידע מצד המערכת, לא יודעים איפה זה קרה בקוד ומה הבעיה (אנו יודעים רק את תוצאת הבעיה). זמן דיבוג ארוך.
    - באגים לא ידועים – אנו לא יודעים כלום ולא נפתור אותן עד שלא יהפכו לשגיאות מאחד הסדרים האחרים.
* **שינוי סוג השגיאה:**
  + באגים לא ידועים:
    - כל שגיאה שהקומפיילר לא מזהה היא באג לא ידוע עד שנריץ את התוכנה, ככל שנריץ אותה יותר פעמים ועל יותר תרחישים יותר באגים מתגלים ע"י מערכת או על ידינו.
      * לשם כך מתכנתים כותבים טסטים אוטומטיים.
      * פתרונות טכנולוגיים Junit
  + ניפוי שגיאות – מיומנות.
    - הדיבאגר עוזר לנו לכך
  + אחת הדרכים הקלות והנפוצות לאתר מקום של שגיאה היא המרת השגיאה לשגיאה מסדר שני – להוסיף לקוד בדיקות שיגרמו לו לזהות מצבים לא רצויים (למשל וידוא תקינות קלט, assert)
  + מעבר לשגיאות קומפילציה
    - הקומפיילר כבר מזהה יותר בעיות מהשאר
    - שימוש בשפה סטטית – כל הישויות והיחסים בין הישויות מוגדרים היטב בזמן כתיבת הקוד (לעומת שפה דינאמית כמו פייתון שבה כל המשתנים והמחלקות מוגדרים בזמן ריצה)
    - השלמת קוד חכמה
* **העשרה:** assert (בעברית: "הנני טוענ\ת כי...") היא פקודה שתפקידה לוודא הנחות שאנחנו מניחים על הקוד כמפתחים. מדובר בהנחות שאמורות להיות בהכרח נכונות, ואם הן לא מתקיימות, משתמע שיש באג בקוד שלנו.
  + היתרון של assert על פני בדיקה רגילה, הוא שבדיקות assert מורצות רק אם אנחנו מציינים זאת מפורשות.
  + בנוסף, assert נבדלת מבדיקה רגילה הוא בבדיקות לזמן דיבוג וכתיבת הקוד, אבל לא בהכרח תרצה שתתרחש בזמן ריצה אצל הלקוח - למשל כי הבדיקה יקרה, ואתה חושב שבמהלך הפיתוח כבר וידאת די הצורך שהיא תמיד מתקיימת.
* **מבוא לחריגות:**
  + טיפול בשגיאות זמן ריצה
  + תוכנית טובה יודעת להתאושש משגיאות כשזה אפשרי (באחריותה לתת תיעוד מאיזה שגיאות היא יודעת להתאושש)
    - כמו כן היא מתאוששת משגיאות במקום המתאים ביותר
  + חריגות – eceptions – הודעה שמשהו השתבש
    - המנגנון הוא ע"י מתודה שנתקלת בבעיה ומחזירה למי שקרא לה שקרתה תקלה.
    - חריגות עומדות מול ערכי החזרה – כלומר במקום ערך החזרה לשיטה נוכל להשתמש בחריגות.
    - איך משתמשים?
      * כשיש בעיה והמתודה לא יכולה להמשיך היא תזרוק חריגה למי שקרא לה.
      * בג'אווה חריגות הן אובייקטים וההודעות שנזרקות חזרה הן אובייקטים אלו.
      * אחרי () של רשימת הפרמטרים של המתודה נוסיף trhows <type> עם סוג החריגות שמחזירה.
      * כאשר נרצה לזרוק שגיאה נכתובthrow new XXException() – נקראת אחרי שזוהתה שגיאה.
        + מפסיקה את המתודה מיידית – אף קוד אחר מהמתודה לא רץ ולא מוחזר ערך
        + רק נזרקת החריגה
      * throws XXException – אומרת שהמתודה יכולה לזרוק שגיאה מסוג זה
        + חלק מהצהרת המתודה
        + חשוב לתעד את זה (JAVADOC - @trows
  + הצד שקרא למתודה הבעייתית ומטפל בחריגות:
    - נשים בה בלוק try{} ובו את הקריאה הבעייתי בפוטנציה
    - אחרי זה נוסיף בלוק catch שתופס את החריגה ומטפל בה – catch(XXXException e) כלומר נצטרך לרשות את הסוג של החריגה שהבלוק תופס
      * אם יכולות להיות כמה חריגות שונות נשים כמה בלוקי catch שונים הנבדלים בסוג החריגה שתופסים
      * קוד זה רץ אם אחת המתודה שהורצה בבלוק try זרקה שגיאה.
      * אנו למעשה מגדירים לו פרמטר מסוג החריגה שתופס ונקבל אובייקט e שיכול להכיל מידע שימושי ורלוונטי –
        + בפרט בעצם הסוג שלו (של החריגה)
  + דרך נוספת לטפל בחריגה היא לזרוק מחדש את החריגה ללא שימוש בבלוק try,cath – נוסיף למתודה הנוכחית את throws XXXException ואז אם תזרק אליה שגיאה זו היא תעביר אותה הלאה למתודה שקראה לה.
  + באופן כללי, כלל אצבע הוא להשתמש בחריגות קיימות שמתאימות לסיטואציה על פני הגדרת חריגה חדשה עם אותה משמעות.
* **חריגות שקופצות מ-main:**
  + לפעמים אנו מריצים תוכנה ורואים שהיא עפה עם פירוט שמכיל את המילה exception
  + זה קורה כאשר הפונקציה הראשית main זרקה חריגה.
  + אמרנו שאם אנו זורקים חריגה היא תיזרק לפונקציה שקראה לה אבל כאשר הmain זורק חריגה אין מי שיתפוס אותה כי אין מי שקרא לו
  + בעצם מי שקרא לו הוא המשתמש ולכן זריקת החריגה תהיה אליו.
* **מוטיבציה לשימוש בחריגות:**
  + נקבל הפרדה ברורה בין החלק שמבצע את המשימה לבין החלק שמתמודד עם השגיאות
  + יותר אלגנטי
  + האפשרות לפעפע שגיאות במעלה מחסנית הקריאה
    - אם רק מתודה מסוימת במעלה העץ יודעות לטפל בשגיאות נרצה לזרוק את השגיאה לשם (מבלי לבדוק כל פעם אם המתודה הצליחה)
  + צריך לזכור להוסיף לכל מתודה איזה סוג חריגה היא יכולה לזרוק ולשים לב אם היא תרצה לזרוק שגיאה שפונקציה אליה קוראת זורקת (במקום לטפל בה)
* **Checked and Unchecked Exceptions:**
  + היררכיה של exceptions:
    - Throwable
    - ממנה יורשים:
      * Error – בד"כ שגיאות מערכת הפעלה
      * Exception שאותו ממשים runtimeerrors (יש כמה סוגים שממשים גם אותו) ועוד סוגים שממשים את exception
  + Unchecked errors – בעיות שנובעות מבאגים בתוכנה שלנו, לא אמורות לקרות אם התוכנה שלנו עובדת טוב
    - יורשות מ-RunTimeException
    - לרוב טעויות של המתכנת - למשל רפרנס שהוא NULLאינדקס לא קיים, לחלק ב-0 וכדומה.
    - מה שמאפיין אותן הוא שהן יכולות לקרות בהרבה תרחישים שונים
    - בגלל זה (שהדברים נפוצים) לא נוכל לצפות שכל מתודה תוסיף את ההצהרה עבור החריגה הכללית והטריויאלית
  + Checked errors – בדר"כ נובעות מקלט לא נכון מהמשתמש
    - יורשות מהמחלקה exception
    - הרבה פעמים ספציפי לתוכנית מסוימת
    - ברמה הטכנית – ברגע שמתודה מחליטה לזרוק חריגה מסוג זה אנחנו חייבים להוסיף את throws statements וכל פונקציה שקוראת לה חייבת או בלוק try/catch או להוסיף גם הצהרת throws
* **ניצול ההיררכיה של חריגות:**
  + סיבה נוספת להשתמש בחריגות היא היכולת לאחד ביחד סוגים שונים של חריגות ולהפריד ביניהם במידת הצורך.
  + כלומר אם יש לנו שני סוגי חריגות שיורשים מאותה מחלקה של חריגה נוכל להגיד שהשיטה זורקת חריגה מהסוג של מחלקת האב
  + הקוד שקורא לשיטה כזו – נוכל לשים בלוק catch לכל אחד מסוגי השגיאות או בלוק catch למחלקת האב
* סיבות עיקריות להשתמש ב-Exceptions:
  + יצירת הפרדה בין הקוד שעושה את הפעולה לבין הקוד שמטפל בשגיאות
  + שימוש ב-Exception כדרך לפעפע שגיאות במעלה מחסנית הקריאות
  + היכולת לאחד ביחד סוגים שונים של שגיאות ולהפריד סוגים אחרים.
* **מתי והיכן כדאי לנו לתפוס חריגות?**
  + תרגום מחרוזת שמכילה מספר למספר עצמו – אם המחרוזת לא מכילה מספר נזרוק שגיאה כי לא נוכל להתקדם בשיטה.
    - מקרה בו המתודה לא מתאימה לטיפול בשגיאה וצריכה לפעפע אותה כי היא לא בתחום אחריותה.
  + אם למשל שיטה מקבלת קלט והוא לא נכון אז היא לא תזרוק חריגה אלא תבקש קלט חדש מהמשתמש כי זה בתחום אחריותה הטיפול בקלט.
  + אם הקלט אינו תקין מי שקרא לפונקציה צריך לטפל בכך.
* **לסיכום:**
  + כאשר קראנו לשיטה שעשויה לזרוק חריגות, נשאלת השאלה: לתפוס או לא?
  + כלל האצבע הוא שתופסים רק אם אחד המקרים הבאים מתקיים:
    - אנחנו יכולים לטפל בחריגה במיקום הזה בתוכנית ולהמשיך בריצה.
    - ברור מעבר לכל ספק שהתוכנית לא יכולה להמשיך, ורוצים להקריס את התוכנה בצורה שקטה עם הודעה אינפורמטיבית.
    - אם אנחנו רוצים להציף הלאה חריגה מסוג אחר.
    - אם אנחנו רוצים לפעפע, אבל לפני כן צריך להריץ קוד שמנקה את מה שעשינו בינתיים.
  + אם אף אחד מאלה לא מתקיים, לתפוס זה רעיון רע כי לא יהיה לנו משהו מיוחד לעשות, ואנחנו לא מאפשרים לשיטות מעלינו (שאולי עבורן אחד המקרים כן תופס) לתפוס.
  + בכל מקרה, אף פעם לא תופסים עם catch ריק (מטאטאים את החריגה מתחת לשטיח).