שיעור 1: שלום עולם, תרגול 1, 2: מחרוזות ומערכים:

מפרש (interpreter): מריץ את הקוד שנכתב בשפת Java. חסרונות:

- מאט את מהירות הריצה.
- גילוי שגיאות רק בזמן ריצה.

מהדר (compiler): מעבד את קוד התוכנית לפורמט נוח יותר - **byte code** הנשמר עם סיומת class. בתהליך הקומפילציה נבדק התחביר ומתגלות שגיאות.

יבילות (Portability): קוד Java חוצה סביבות, כלומר קוד שקומפל יוכל לרוץ על בכל מחשב ובכל סביבה בה מותקן מפרש לשפה.

המכונה המדומה (Java Virtual Machine – JVM): מפרש הנכתב עבור כל סביבה (פלטפורמה), ומכאן שה-JVM כן תלוי סביבה. את ה-JVM לא כותב המתכנת.

חבילת ערכת פיתוח (Java Development Kit – JVM): כוללת JVM, קומפיילר, דיבגר וכו'.

סביבת פיתוח שלובה (Integrated Development Environment – IDE): סביבה המשלבת כלי פיתוח עצמאיים: עורך, סייר, קומפיילר, JRE, וכו'. למשל: אקליפס.

נקודות חשובות:

- שמות משתמשים אינם חלק מחתימת המתודה.
- הערות: שלושה סוגי הערות:

```
// text – עד סוף שורה
/* text */ - הערה על מספר שורות
/** text */ - javadoc – הערת תיעוד
```

סוגי טיפוסים: יסודיים (8 סה"כ), טיפוסי הפניה (מחלקות). יתרונות להגדרת טיפוס מראש:

- יעילות זמן חישוב (שלמים לעומת עשרוניים).
- חסכון בהקצאת זיכרון.
- סוג מגדיר אילו פעולות ניתן לבצע עליו.

תהליך ההשמה הוא אימפרטיבי, חישוב נעשה מימין לשמאל.

- Char: ייצוג ע"י תו בין ' ל-', או ע"י מספר לפי טבלת ה-unicode. '\n' – שורה חדשה, '\t' – טאב.

המרת טיפוסים (Casting):

- בטיחות ההמרה מתייחסת לסוג הטיפוס ולא לערכו.
- המרה בטוחה נקראת widening.
- המרה מפורשת (explicit casting) – "לקיחת אחריות" על ההמרה גם אם לא בטוחה, narrowing.
- כל השמה של מספר מהצורה: X.X (עם נקודה צפה) מתפרש כ-**double** ולא כ-float.

- משפט (statement) – מבצע פעולה; ביטוי (expression) – מבנה שניתן לחשב את ערכו. לרוב משפט לא יחזיר ערך, אך יש שכן, למשל ביטוי השמה.
- השמה: x++ - מחזיר x ולאחר מכן מקדם; ++x – מקדם את איקס ומחזיר אותו עם ערכו החדש.

- השמה מתבצעת מימין – x=y=z קודם z ל-y, אח"כ y ל-x; חיבור, חיסור וכו': משמאל לימין.

משתני הפניה:

- משתנים יסודיים תופסים מקום מוגדר מראש ב-stack, ומשתני הפניה מכילים כתובת לזיכרון ב-heap.
- יצירת מערך באופן: int[] arr = new int[5]; ייאתחל את חמשת אברי המערך ל-0, בולייני ל-false.
- אם שתי הפניות מצביעות לאותו עצם, אין עותק נוסף שלו, אלא ההצבעה היא לאותו מקום בזיכרון.
- המרת מחרוזת לטיפוס יסודי: Integer.parseInt("<number>") (או Float או כל אחר).

שיעור 2: מבני בקרה, תרגול 2: מבני בקרה, תרגול 3: מבני בקרה:

לולאות:

- אופרטור התנאי <Boolean expression> ? <t-val> : <f-val> **שונה** ממשפט התנאי if בכך שאינו משפיע על זרימת התוכנית. אופרטור התנאי מחזיר ערך כלשהו.
- Switch מקבל טיפוס שאינו טיפוס מניה וה-case משווים את הערך אליו. עבור כל תנאי שאינו ב-case ניתן בלוק (בד"כ ימוקם אחרון) default. כדי שאחרי התקיימות case נצא מהביטוי, יש לשים **break;**, אחרת נמשיך לבדוק את שאר ה-caseים.
- while, do-while נבדלים בכך שהשני מבצע ריצה אחת של ה-statement לפני בדיקת התנאי.
- בלולאות ניתן למקם בחלקי האינקמנטיזציה והאיתחול יותר ממשתנה אחד, ע"י הפרדה בפסיקים.
- בתוך לולאה ניתן להשתמש ב-break לעצירה מוחלטת של הלולאה או **continue** – ליציאה מהאיטרציה הנוכחית של הלולאה ומעבר לבאה.

שרותי מחלקה (static methods):

- קריאה למתודה ממחלקה אחרת מצריכה שם מלא: <class name>.<method name>

העמסת שירותים:

- ניתן לכתוב פונקציות בעלות אותו שם, כל עוד **החתימה שלהן שונה**. שונות תהיה בטיפוס ו/או סוגי ומספר הארגומנטים, אך **לא בשמות משתנים**.
- יתרונות:
  - נוחות
  - ערכי ברירת מחדל: אפשרות להעמיס מתודה קצרה עם ערכי ברירת מחדל פנימיים, עבור ערכים שחוזרים על עצמם.
  - תאימות לאחור: במידה וכבר יש מתודה ומשתמשים בה גם במחלקות אחרות, לשנות אותה יפגע בקימפול של קוד משתמש, לכן ניתן להעמיס גרסה חדשה של המתודה כנדרש. בעיה: שכפול קוד.
- הקומפיילר יחליט איזו העמסה לבחור עבור קלט מסויים, ואם אין העמסה מדוייקת יבצע המרה (אם יוכל), ואם יש תאימות לשתי העמסות – יחזיר הודעה על אי בהירות.
- לשכפול מס' כלשהו של ארגומנטים יש פתרון של תחביר: <method>(<type> ... <array-name>).

משתני מחלקה:

- משתנים גלובליים למחלקה (או: שדות מחלקה) יוגדרו עם **static**:
  - מוגדרים בכל המחלקה.

- מוגדרים בזיכרון כל עוד המחלקה בשימוש.
- מאותחלים לפי ערך ברירת מחדל (גם בלי שניתן להם ערך ספציפי).
- ישבו ב-**Heap** ולא ב-Stack.

- משתנה **final** ניתן לאתחול פעם אחת בלבד, כל השמה אח"כ תזוהה כשגיאת קומפילציה.

שגיאות קומפילציה מול שגיאות זמן ריצה:

- שגיאות הידור: נתפסות בעת ניסיון להפוך קוד ל-bytecode.
- שגיאות זמן ריצה: שגיאות שלא ניתן לתפוס בעת הידור.

שיעור 3: דיני חוזים, תרגול 3: חוזים:

מבני בקרה:

- כדי לקצר ניתן להשתמש ב-import: import java.util.someClass מאפשרת שימוש ישיר ב-someClass.
- שימוש ב-*.java.util ייתן את כל תתי המחלקות במקום המתאים ל-* **אך לא בצורה רקורסיבית**, כלומר לא את **כל** תתי ותתי-תתי וכו' המחלקות שבמקום *.
- static import: מאפשר ייבוא של משתנה של מחלקה / מתודה. גם כאן ניתן להשתמש ב-*.
- אין צורך לייבא מאותה **חבילה**, או את java.lang.
- **CLASSPATH:** משתנה סביבה המכיל את שמות התיקיות המכילות מחלקות בהן משתמשים. ניתן להוסיף ארכוב מחלקות מפורמט jar ל-classpath.

API (Application Programming Interface) ו-javadoc:

- ה-javadoc יוצר API אוטומטית בפורמט HTML, והיא חלק מה-JDK.
- עיצוב לפי חוזה: תנאי קדם ותנאי אחר.
- לא בודקים תנאי קדם! אחרת הם כבר לא תנאי קדם.
- ניתן להחזיר לאחר ביצוע מתודה תוצאה חזקה מתנאי האחר שהוגדר.
- **משתמר:** תנאי שצריך להתקיים לכל אורך ריצת התוכנה, והוא מגדיר את נכונות התוכנה. הוכחת משתמר היא על פי כללי המתמטיקה.

נראות פרטית (private visibility):

- מאפשר גישה למשתנה רק מתוך המחלקה בה הוגדר.
- תיחום באמצעות נראות פרטית מכונה **הכמסה** (encapsulation).

משתמר הייצוג:

- משתמר הקשור לייצוג הפנימי של משתמר כלשהו של התוכנית. למשל:

```
/** @inv getCounter() == #calls for m()
*   @imp_inv counter == calls for m()
כאשר הייצוג הפנימי במימוש של הוא משתמר הייצוג -      */
```

- קיימים משתמרי ייצוג גם עבור תנאי בתר.

נראות:

- כאשר לא מצויינת דרגת נראות (public, private), ברירת המחדל היא package.

מחלקות:

- בנאי: פונקצית אתחול המקצה למופע מחלקה (עצם) מקום על ה-Heap.
- Heap – תכיל משתנים גלובלים ועצמים, לא תלוי במתודה הנוכחית שרצה.
- Stack – משתנים מקומיים וארגומנטים.
- **שרותי מופע** בניגוד ל**שרותי מחלקה** (static methods) עובדים על עצמים מסויימים.
- **משתני מופע** בניגוד ל**משתני מחלקה** (static fields) הם שדות בתוך עצמים ויחודיים להם.

שיעור 4: מחלקות ומנשקים, תרגול 4: מחלקות ועצמים:

שירותי מחלקה:

- בנאי: פונקציית האתחול.
- שירותי מחלקה: מסומנים static
- שירותי מופע: לעצם ספציפי, גישה עם אופרטור "."
- סוגי שרותים: שאילתות, פקודות ובנאים.

מצב מופשט ופונקצית הפשטה:

- תיאור הטיפוס בצורה מדויקת, פשטנית ומתמטית.
- יצירת חוזה המחלקה בהתאם לצורה זו
- פונקצית ההפשטה: מיפוי אצמים קונקרטים למצב המופשט.
- בחוזה:

@abst <abstract description> - בראש המחלקה

@abst AF(this) == <description> - תיאור מצב לאחר הפעלת השרות

- פונקצית ההפשטה אינה חח"ע (שכן הרבה עצמים יכולים להיות באותו מצב מופשט).
- שימוש במשתמרי ייצוג להוכחת נכונות גם לגבי המצב המופשט.

ניראות תיעוד: javadoc תומך בדרגות ניראות שונות (למשל private לא יופיע בתיעוד הציבורי).

מנשקים:

- במנשק יוגדר חוזה ומצב מופשט של הטיפוס.
- יתרונות למנשקים:
  - צמצום תלות בין קוד הלקוח למימוש הטיפוס – גמישות לקוד הלקוח.
  - חופש פעולה במימוש הטיפוס.
  - גמישות בשינויים עתידיים במימוש – גמישות ליוצר הטיפוס.
- **לא ניתן** להגדיר במנשקים בנאים ומתודות סטטיות.

שיעור 5: מנשקים, תרגול 5: מחלקות ועצמים חלק ב':

תבנית עיצוב Factory:

- עבור מחלקת אם המשתמשת בטיפוס כלשהו, נרצה לצמצם נראות למנשק, אך אם קיימת תלות גם ככה בין מחלקת האם למחלקה המממשת את המנשק, נגדיר מחלקה חדשה Factory שתייצר כל פעם אובייקט מהמחלקה המממשת.
- מחלקת האם תשתמש במחלקת המפעל לייצור מופעי המנשק, וישנה תלות בין מחלקת המפעל למימוש.

אופציה נוספת: **שימוש במנשקים**: כדי להפחית תלות בין מחלקת האם למחלקה המממשת, נשתמש במחלקת האם רק במופעים של המנשק, כך שלא תהיה גישה לשדות פנימיים או שירותים שלא הוגדרו במנשק.

- לטיפוס מסוג מנשק ניתן לבצע השמה של כל טיפוס מחלקה המממשת את המנשק, אך לא הפוך.
- **פולימורפיזם (רב צורתיות):** לכל מנשק יכולים להיות מספר מימושים, וקוד המשתמש במנשק לא צריך להתחשב במימוש אחד ספציפי, וכך לשירותים המופיעים במנשק צורות רבות – למימושים שונים. הקומפיילר מוודא שלכל מחלקה מממשת מנשק יש את כל השירותים המוצהרים במנשק. התאמת הפונקציה המתאימה שתקרא בזמן ריצה נקרא **dynamic dispatch.**
- ניתן להגדיר מערכים של טיפוס מסוג מנשק.
- **ריבוי מנשקים:** מחלקה יכולה לממש מספר מנשקים: public <class_name> implements I1, I2, I3

שיעור 6: מחלקות פנימיות, תרגול 6: מבני נתונים מוכללים, תרגול 7: מחרוזות:

מחלקות ושירותים גנריים (מוכללים):

- הוספת <T> בצמוד לשם המחלקה.
- טיפוס בתוך המחלקה מוגדר כ-<calss_name><T>
- ניתן להשתמש ב-T כטיפוס גנרי.
- הטיפוס הקונקרטי חייב להיות מסוג הפנייה, ולא יכול להיות פרמיטיבי (למשל Integer ולא int), לכן נשתמש בטיפוסים עוטפים:
  - לכל פרמיטיבי קיים טיפוס עוטף והוא immutable, למשל Integer הוא int ב"עטיפה" ולא ניתן לשנות את תוכנו ברגע שנוצר.
  - קריאת הערך המוחזק בטיפוס עוטף (unboxing) ע"י <var>.<var-type>Value.
  - בג'וואה 5 נעשה boxing, unboxing אוטומטי.

מחלקות פנימיות (מקוננות – Nested classes):

- מבנה תחבירי המבטא היכרות אינטימית בין מחלקות. למשל, טיפוס הנועד לצורכי שימוש טיפוס אחר, הראשון ימומש כמחלקה אינטימית בתוך האחר.
- רמת ניראות דיפולטית היא רמת המחלקה – עצמים מאותה מחלקה יכולים לגשת לעצמים אחרים מאותה מחלקה, גם לשדות פרטיים.
- למחלקה פנימית גישה לשירותי המחלקה העוטפת, ויש לה שדה הפניה אוטומטית לעצם מהמחלקה העוטפת.
- אם יש שמות משתנים חוזרים, הדיפולטי יהיה של **המחלקה הפנימית**, וכדי לגשת לזה של המחלקה העוטפת נשתמש בשם מלא: <outer_class>.<var_name>.
- יצירת מופע של מחלקה פנימית:
  - בתוך המחלקה העוטפת: יצירה רגילה.

- לא בתוך המחלקה העוטפת: יצירה ע"י outerObej.new innerObj – כיוון שאין משמעות לאובייקט פנימי שלא מקושר למופע של האובייקט החיצוני.

- **מחלקה פנימית סטטית:** מחלקה פנימית שאינה מקושרת למופע של המחלקה החיצונית. קריאה לה ע"י: outerClassName.InnerClassName.

אם מחלקה פנימית סטטית אינה public, צריך ליצור לה getter. למשל:

Outer.Inner obj = new Outer.Inner(); - error

Outer.Inner obj = new Outer.getInner(); - ok - כאשר הגטר הוא ציבורי במחלקה העוטפת

מחלקות פנימיות בתוך מתודות:

- תחום ההיכרות של המחלקה הפנימית רק בטווח המתודה.
- יכולה להשתמש במשתני המתודה **רק אם הם final.**
- ניתן גם ליצור מחלקות פנימיות אנונימיות.

קומפילציה של מחלקות פנימיות יניב קובץ class נפרד, ששמו <outer_name>$<inner_name>.class, ועבור אנונימיות: <outer_name>$1.class וכן הלאה (מספור).

Iterator Design Pattern:

- הפשטה של מעבר על מבנה נתונים, מאפשר סריקת מבנה נתונים ללא הכרת המימוש הפנימי.
- איטרטור הוא <u>מנשק</u> שחייב לממש:
  - hasNext() – בודק האם הגענו לסוף.
  - next() – גם מחזיר את הנתון וגם מקדם.
  - remove() – הסרה של נתון (אופציונלי, לרוב ימומש מימוש ריק – { }).
- כל מחלקה שיש לה איטרטור: implements Iterable<T> (אם היא גנרית עם <T>).
- פונקציית החזרת האיטרטור נראית כך:

```
Public Iterator<T> iterator(){
    Return new iteratorImplementation<T>(args);}
```

- מקובל להכניס את מחלקת iteratorImplementation<T> בתור מחלקה פנימית של הטיפוס.
- Collection<E> ממש את Iterable<E>, ו"ממש" היא 75% מ'.
- ניתן לבצע for-each על איטרטור.

Strings:

- מחרוזת היא immutable, לכן ניתנת להצבעה ע"י יותר ממשתנה אחד.
- כל המחרוזות השונות נכנסות ל-string pool, שם נמצאות כל עוד מצביעים עליהן.
- StringBuffer – סוג של מחרוזת mutable. ניתן לבצע עליה Insert, append.
- קיים גם StringBuilder, אבל לא נראה לי שזה חשוב.

שיעור 7, 8: הורשה, תרגול 7: בנאים, תרגול 8: תבניות הורשה ומימוש:

יחסים בין מחלקות:

- **מכלול (Aggregation):** יחס המבטא הכלה, למשל collection: מכיר את רכיביו, אך הם לא מכירים אותו, לרכיבים אפשרות קיום עצמאית ללא היכללות במיכל.
- **הרכבה (Composition):** הכלה בה לרכיבים **אין** יכולת קיום ללא המיכל, למשל טיפוסים המוגדרים במחלקה פנימית בתוך טיפוס המיכל (כמו חדרים בבית, שכן לחדר אין קיום לבדו).
- **יחס is-a (Generalization):** כאשר מחלקה אחת היא סוג של מחלקה אחרת.

← הורשה:

- מחלקה יורשת מקבלת את כל תכונות המורישה.
- ניתן לרשת **אך ורק ממחלקה אחת** (לעומת מימוש, שכן מחלקה אחת יכולה לממש מספר מנשקים).
- בנאים **לא עוברים** בירושה, ומתודות סטטיות **לא עושות אוברריידד** בירושה.
- אין ניראות שדות private.
- ממחלקה שתוגדר **final** לא ניתן יהיה לרשת.
- ישנה הורשה בין מנשקים.

אתחולים ובנאים:

- כל בנאי של מחלקה שלא יורשת משום מחלקה אחרת, מכילה בנאי (נסתר) של object כיוון שכל מחלקה מרחיבה את object.
- בנאי של מחלקה יורשת יכיל קריאה לבנאי של המחלקה ממנה יורש ע"י קריאה ל-()super עם הנתונים המתאימים. הקריאה הנסתרת לבנאי של object תהיה ריקה.
- **דריסה:** מחלקה יורשת יכולה לדרוס מתודה מהמורישה ע"י שימוש בחתימה זהה. כדי לקרוא למתודה הנדרסת נשתמש ב: super.methodName. בד"כ נוסיף לפני @Override

טיפוס סטטי ודינאמי:

- טיפוס של עצם הוא טיפוס הבנאי לפיו נוצר העצם והוא בלתי ניתן לשינוי ברגע שנוצר.
- טיפוס סטטי: הטיפוס שמוגדר בהכרזה על ההפניה (מקדם שם המשתנה) – יכול להיות מחלקה או מנשק.
- טיפוס דינאמי: טיפוס העצם המוצבע, והוא נגזרת של הטיפוס הסטטי. למשל: PolarPoint של IPoint.

ניראות Protected:

- מאפשרת גישה לאורך שרשרת הירושה, וברמת החבילה (בשפות מונחות עצמים אחרות ניראות זו לא כוללת את רמת החבילה).

סיכום ניראות וירושה:

	גישה מאותה מחלקה	גישה מאותה חבילה	גישה מתת-מחלקה	אחרים
Private	כן	לא	לא	לא
Package (default)	כן	כן	לא, אלא אם המחלקה ותת-המחלקה באותה חבילה	לא
Protected	כן	כן	כן, גם אם המחלקה ותת-המחלקה לא באותה חבילה	לא
Public	כן	כן	כן	כן

למה protected בהורשה	למה private בהורשה
• מקיים יחס is-a, הגיוני שיהיה בעל אותן זכויות • היורש מכיל בתוכו את המוריש, לכן צריך גישה פשוטה ונוחה	• הסתרת המימוש לשמירה על שלמות המידע – גם מצאצאים • יורש עלול לשבור חוזה מוריש, ולשבור את התוכנה אצל הלקוח • יורש הוא סוג של לקוח, עיקרון הסתרת המידע צריך לחול עליו

Clone, deep_clone:

- Clone יוצר מצביע אל אותו אובייקט.
- deep_clone יוצר אובייקט חדש עם שדות חדשים, אך השדות החדשים מצביעים על אותם אובייקטים שהשדות הישנים מצביעים (כלומר לא deep_clone רקורסיבי).

Template method design pattern: מחלקה מופשטת - abstract class:

- לא חייבים לממש מתודות נורשות, ניתן ליצור חתימות של מתודות חדשות בלי מימוש.
  - מתודות ממומשות נקראות <u>אלגוריתם כללי</u>, אינן תלויות במימוש הספציפי (כמו החלפת top במחסנית – צריכה להשתמש ב-push, top, מבלי לדעת כיצד ממומשות).
  - מתודות אבסטרקטיות המועברות הלאה נקראות <u>hooks</u> או <u>callbacks</u>.
- **לא ניתן ליצור מופע של מחלקה מופשטת**.
- נועדה למנוע שכפול קוד.
- אפשר לוותר על הצהרות שמתקבלות מהמנשק ולא ממומשות.
- המחלקה המופשטת באה לחסוך לספק, לנו, ואילו מנשק – ללקוח.
- אקליפס מאפשר שכתוב מבני (refactoring) הנקרא Extract superclass ליצירת מח' אבסט' המאחדת מספר מחלקות יורשות, ומאחדת אוטומטית את הקוד שניתן לאחד.

Adapter Design Pattern:

- תיאום בין מנשק למחלקה שלא מממשת אותו, ע"י מחלקת תיאום. במחלקת התיאום יהיה שדה:
  <class-name, e.g: java.util.Random> <var-name, e.g: r>;
  המחלקה תורכב ממתודות מימוש המנשק, תוך התאמה בין r לנדרש.
- ניתן לעשות זאת על יותר ממחלקה אחת.

Bridge Design Pattern:

- פתרון לבעיית צורך בירושה מרובה ע"י שימוש בהכלה עם האצלה.
- למשל, ניצור שני מנשקים לטיפוס, כאשר מחלקה המממשת את הראשון תקבל כשדה (בבנאי) מופע של מימוש המנשק השני, ועל מימושו תוסיף לוגיקה כלשהי (דוגמא במצגת: מחסנית המממשת ע"י LastModifiedStack – לוגיקה של שמירת שדה תאריך שינוי אחרון, המקבלת בבנאי מופע של מימוש ArrayList של מחסנית – לוגיקה מקבילה שאינה מתנגשת עם הראשונה).

טיפוסי זמן ריצה:

- בשל הפולימורפיזם, לא ידוע הטיפוס המדויק של עצם, כלומר הטיפוס הדינמי, וכך נחסמות אפשרויות שניתנות לטיפוסים דינמיים ספציפים. בשביל לפתוח אפשרויות אלו נשתמש ב-casting.
- סוגי המרות:
  - Downcast: המרה למטה, בפציפיקציה של הטיפוס. יכול להוות בעיה.
  - Upcast: המרה למעלה למחלקה או מנשק, לא מהווה בעיה (פשוט מאבד מידע).

  המרה תתבצע כמו בטיפוסים פרמיטיביים: (Type-to-cast-into) <Expression>
- בדיקות טרום-casting:
  - מתודת getClass() מחזירה את המחלקה של הטיפוס. למשל הבדיקה:

    shape.getClass() == Polygon.class
  - Instanceof: (shape instanceof Polygon) – בודק האם העצם הנבדק הוא יורש מהטיפוס, או מממש את הטיפוס או מסוג הטיפוס עצמו.

טיפוס כללי:

- ברגע שנוצר טיפוס ספציפי מסוג טיפוס כללי, למשל FCStack<String> **אינו סוג של** FCStack<Object>. הערות חשובות:
  - אם יצרנו טיפוס כללי, והשמנו בו טיפוס ספציפי, למשל מערך של object שהכנסנו אליו מערך של String, אם נמלא אותו בערכים שאינם String – **יעבור קימפול אך ייזרק בזמן ריצה.**
  - כמו כן **לא ניתן ליצור מערכים של טיפוס גנרי (טעות קומפילציה)**, אך כן ניתן לעשות casting למערך מטיפוס גנרי.
- **טיפוס נא:** אם ישנו טיפוס someType<T>, ניתן ליצור ממנו טיפוס נא:

  בלי לציין את הסוג, שקול ל – someType var = new someType():

  הסוג הוא הגבול העליון מבחינת טיפוסים, לרוב אובג'קט - someType<?> var = new someType<Object>

  במקרה זה **כן ניתן לבצע השמה (כן מתקמפל ועלול ליפול בריצה) של הטיפוס הנא לטיפוס ספציפי.**
- השמת גבול לטיפוס כללי: public class someClass<T **extends Comparable**> - למשל, מגביל את T לקבל טיפוסים שממממשים Comparable.
- בזמן ריצה אין זכר לפרמטר הטיפוס הכללי, ולכן לא ניתן לקרוא לבנאי לפי הפרמטר הכללי.
- כדי לכתוב שירות המקבל פרמטר כללי, לא נוכל להשתמש ב-<Object>, כי כפי שנאמר לעיל, נקבל שגיאת זמן ריצה בניסיון השמה של כל מה שאינו בדיוק Object. לצורך כך נשתמש ב**ג'וקר**:

  כך המתודה מוגדרת לקבל כל מיני טיפוסים עם מכנה משותף כלשהו - <? Extends (upper limit)>

  השמת חסם תחתון לטיפוס המתקבל - <? Super (lower limit)>

ירושה וחוזים:

- שמורת כל מחלקה צריכה להיות שווה/חזקה משמורת מחלקת הבסיס/המנשק ממנה יורשת/אותו מממשת.
- תנאים אפקטיבים: מכלול התנאים לאורך עץ הירושה. בכל שלב יוגדרו בחוזה התנאים החדשים בלבד.
- **השמורה האפקטיבית**: היא שמורה המורכבת משמורות כל המחלקות בשרשרת הירושה המקושרות ע"י AND (שכן שמורות כל המחלקות בדרך צריכות להתקיים). מהגיון זה, אם בדרך למחלקה כלשהי אין שמורה, ינתן לה ערך ברירת מחדל אמת.
- **תנאי קדם אפקטיבי:** תנאי קדם של מחלקה יורכב מ-OR של כל תנאי הקדם של הוריה, שכן החזק מתנאי הקדם יהיה של מחלקת הבסיס (כי לא יכול להיות שמחלקה בשרשרת תצמצם את טווח הפעולה של הוריה, תנאי הקדם שלה יהיה שווה / חלש משל הוריה). מהיגיון זה, אם למחלקה בדרך אין תנאי קדם, כדי שלא תתרום דבר ל-OR, ינתן לה ערך ברירת מחדל שקר.
- **תנאי אחר אפקטיבי:** שרשרת AND של תנאי האחר של שרשרת המחלקות בדרך, כאשר ברירת מחדל למחלקה בלי תנאי אחר הוא ערך אמת (שכן במורד עץ הירושה תנאי האחר נשמר או מתחזק).
- **עוד חוקים:**
  - במורד עץ ירושה למתודה דורסת מותר להרחיב נראות אך לא לצמצמה.
  - במורד העץ מותר למתודה דורסת לצמצם את טיפוס הערך המוחזר אך לא להרחיבו.
- תנאי קדם מופשט: תנאי קדם של מתודות אבסטרקטיות או שנדרסו; הוא נשאר ללא שינוי, גם אם תנאי הקדם הקונקרטי שונה ממנו.
- סיכום: ירושה צריכה לקיים שני תנאים: יחס is-a ועקרון ההחלפה.

שיעור 9: חריגים:

- **Checked exceptions**: תנאים העשויים להתקיים במהלך ריצה תקין, מקרי קצה. מיוצג ע"י Exception
- **Unchecked exceptions**: בעיות חמורות היוצגות ע"י Error, או באג בתוכנית המיוצג ע"י RuntimeException.
- כל הנ"ל הם טיפוסים שמרחיבים את Throwable, ובחתימת המתודה נוסיף:

  throws <list of exceptions>

- לעתים נרצה להגדיר תנאי קדם חלש, מהסיבות הנ"ל:
  - **חוסר שליטה:**

    חוסר שליטה בגלל בו זמניות: למשל גישה לקובץ, שמרגע בדיקת קיומו עד הפעלת הפעולה עליו הוא עלול להימחק (לכן צריך לתת תנאי קדם חלש יותר, שכן תנאי הקדם שאף פעולה לא תיעשה על הקובץ היא בעייתית).

    חוסר שליטה בגלל פרוטוקולים: העצם מושפע מפרוטוקול (מהעולם החיצון) ולכן מאבדים שליטה.
  - **קושי לבדוק את התנאי:** לפעמים בדיקת התנאי עבור הלקוח תיקח כאורך ביצוע המתודה, וכבר לא שווה להריץ אותה... לכן נדרש לתנאי קדם חלש יותר.

תנאי צד:

- במקום תנאי קדם, מוכנס תנאי צד: שבמידה וחל לא מוחזר הערך לפי תנאי האחר, אלא נזרק חריג.
- בלוק try-catch: את הקוד הבעייתי נעטוף בבלוק try ואם יזרק חריג, יתפס באחד מבלוקי ה-catch בהתאם ל-exception המתקבל. חיפוש handler יתבצע החוצה עד main. אם יש כמה בלוקי catch מתאימים, יתבצע הראשון מביניהם בסדר ההופעה בקוד.
- בלוק finally: אם נזרק תנאי צד, חובה לבדוק שהמשתמר מתקיים, במידה ושונה בדרך. בלוק ה-finally יחזיק את מה שיתקיים בכל מקרה אחרי ה-try-catch.
- חוזה לא חייב להגדיר את תנאי הצד, אך רצוי.

בהצהרת ...throws אין חובה להצהיר על unchecked Exceptions כיוון שאלו בעיות חמורות בסביבה/בתוכנית, ואין משמעות לאחר שקרו לקיום המשתמר שכן התוכנית צריכה להיעצר.

יצירת חריגים חדשים:

- חריגים הם אובייקטים היורשים מ-Throwable או צאצאיו. סיבות שחריגים הם עצמים:
  - שניתן יהיה להחזיר בתוכו מידע שיכול להועיל לתיקון הבעיה.
  - שנוכל להודיע למשתמש על החריג.
  - שיכיל מידע שיאפשר התאוששות (נדיר).
- כל חריג יכיל מינימום: בנאי ריק, בנאי שמקבל מחרוזת, ומתודת getMessage המחזירה מחרוזת.

הורשה:

- לכל מחלקה יורשת / מתודה דורסת **אסור** להרחיב את החריגים הנזרקים, אך כמובן ש**מותר** לצמצם את מספרם או לזרוק חריגים היורשים מהחריגים שירשה , כלומר להקל בסה"כ על הלקוח.

תרגול 9: I/O:

זרמים:

- זרם הוא כמו צינור להעברת מידע. כל זרם נפתח אוטומטית ברגע שנוצר. יש זרמי קריאה וכתיבה:
  - קריאה: InputStream לקריאת בתים, Reader לקריאת תווים.
  - כתיבה: OutputStream, Writer.

  כל הנ"ל הן מחלקות אבסטרקטיות.
- System.in, System.out, System.err – זרמים סטנדרטים לקלט InputStream, פלט ושגיאות (PrintStream).
- Wrappers – מקבלים בבנאי זרם ומוסיפים עליו אופציות. Scanner למשל מחלקת זרם ל-tokens ומאפשרת להוציא את ה-int הבא, ה-char הבא, שורה וכו'. ניתן לשנות לה delimiter.

מחלקה File: אין מה לכתוב את כל הפקודות.

Object serialization:

- Object serialization – מאפשר שמירת וקריאת אובייקטים מזרם בתים.
- מנשק Serializable – מנשק ריק, רוב המחלקות הן Serializable, ותתי מחלקות של מחלקות Serializable הן גם כאלו.

שיעור 10: GUI, תרגול 10: GUI:

- כל דבר ב-GUI הוא widget.
- עצמים ודברים חשובים:
  - Display – מייצג את המסך.
  - Shell – עצמים של חלונות, ההורה הוא המסך.
  - Event loop – לולאה הרצה וממתינה לפקודות.
- יש לשחרר משאבים גרפיים בסוף שימוש, כגון display.
- במתודת יצירת ה-shell נגדיר:
  - ניצור אובייקט מסוג Shell, נגדיר את הכותרת שלו.
  - Shell.setLayout – הגדרת סוג התבנית: עמודות, שורות...
  - לכל אובייקט שניצור ונשים ב-shell, נגדיר לו גם Layout ע"י setLayoutData.
  - מאזינים:
- לכל רכיב ניתן להוסיף **מאזין** ע"י מחלקות אנונימיות (observer design pattern), דוגמא:

Button.addSelectionListener**(** – מוסיף מאזין ללחיצה על הכפתור

new SelectionAdapter(){ – adapters - מחלקות שיודעות כאשר האירוע לו מאזינים קרה, אך לא מבצעות כלום. מהם אנחנו נירש ונדרוס את המתודה widgetSelected.

```
Public void widgetSelect(SelectionEvent e){
        <do something>
}
});
```

- כאשר קורה אירוע, כל מאזין נודע על כך, ומשחרר את האדפטר המוגדר. יכולים להיות כמה מאזינים לאירוע. ניתן גם להשתמש במאזין לא ספציפי – new Listener() – ובמחלקה האנונימית תהיה המתודה public void handleEvent(Event e) שתבצע משהו (עבור כל event שתקבל).
- <u>הערה</u>: ניתן לממש מאזינים גם בדרכים אחרות, כגון מחלקה יורשת מאובייקט שתממש listener, או הכלה של כפתור במחלקה נפרדת (האצלה), או מחלקה פנימית שאינה אנונימית.
  - shell.pack – פורש את ה-shell.
  - shell.open – פותח את ה-shell.
- פריסה: פריסה מתבצעת מלמעלה למטה, כלומר פריסת עצמים בתוך אובייקט תתבצע לאחר שגודל האובייקט נקבע.
- Composites: מאפשרים מודולריות של פריסה.

שיעור 11: מקביליות ורשתות:

תקשורת בין מחשבים:

- פרוטוקול תקשורת: מתאר את תחלופת המידע בין מחשבים. המידע אינו תלוי שפת התכנות בה נכתבו התוכניות המדברות באמצעות פרוטוקול, וישנם תקנים סטנדרטיים לפרוטוקלים (ציבוריים).
- לכל מחשב מחובר לאינטרנט כתובת IP (4 מספרים X 256 אפשרויות), ולכל תוכנית הרצה יש 65,536 אפשרויות לשלוחה - Port.
- תקשורת למחשב תיעשה ע"י פניה לכתובתו עם הודעה מתאימה לפי הפרוטוקול.

תקשורת ב-Java:

- שימוש במחלקה **Socket** (שקע) לתקשורת עם תוכנית מחשב אחרת. שליחת מחרוזת בין מחשבים:

Server	Client
ServerSocket s = new ServerSocket; - יצירת שקע תקשורת Socket clientSocket = s.accept(); - ממתין לקבלת מידע BufferedReader in = new BufferedReader(new InputStreamReader (- קריאת הזרם clientSocket.getInputStream())); String input = in.readLine; - שמירה במחרוזת	Socket s = new Socket("www.soft1.co.il", 7); - יצירת שקע תקשורת PrintWriter out = new PrintWriter(s.getOutputStream(), true); out.println("hello there!"); - הדפסת זרם לשקע התקשורת

תכנות מרובה חוטים (Threads):

- חוטים ב-Java הם אובייקט מסוג **Thread**.
- חוט מריץ מתודה עם חתימה קבועה: public void run(), פרט לראשי המריץ את main()
- מחלקה המממשת את run() ממשת את Runnable.
- Thread הוא עצם שהועבר אליו מופע של מחלקה המממשת את Runnable.
- מספר חוטים יכולים לרוץ על אותו אובייקט במקביל, ולחלוק מידע.

חסרונות לשימוש ב-sockets:

- <u>פרוטוקול לא סטנדרטי</u>: היישום מגדיר פרוטוקול משלו, וכדי להשתמש בתוכנית לקוחות יצטרכו להתקין clients שלה.
- <u>הטכנולוגיה תופסת חלק נכבד מהקוד</u>: לעומת הלוגיקה העסקית (התופסת חלק קטן בקוד).

דפדפנים:

- דפדפנים מחלקים את ה-URL שהוכנס לחמישה חלקים:
  - Protocol – http:// ברירת המחדל.
  - Host – www. … .com/
  - File - /<path>/index.html
  - Port: 80 ברירת המחדל של http.

עבודה עם פרוטוקול http ב-Java:

- שימוש במחלקות הנקראות servlets המממשות לוגיקות עסקיות אותן נטען לשרת - ה-web container.
- השרותון הנשלח מקבל את כל מה שצריך מהשרת, עליו לממש רק את doGet – שהוא hook.
- כתיבת שרותון:
  - <servlet_name> **extends HttpServelt**
  - בנאי ריק
  - Override לפונקצית doGet המקבלת HttpServletRequest, HttpServletResponse
- השרותונים הם סינגלטון, לכן אם כמה לקוחות ניגשים לאותו שירותון, הם מופנים לאותו עצם בזיכרון.

<u>פעולות השרת HttpServer</u>:

- יצירת שקע והמתנה ללקוחות.
- עבור כל הודעה חדשה שמקבל:
  - קרא שורה ראשונה בהודעה.
  - אם זו הודעת GET:
    - ניתוח המחרוזת לחילוץ שם השרותון המבוקש עם שמות המשתנים וערכיהם.
    - קבל הפניה לשרותון המתאים.
    - צור את מחלקות העזר (Request, Response).
    - אם השרותון נמצא, קרא ל-doGet על השרותון ומחלקות העזר, אחרת הצג שגיאה.

תרגול 11: Enumerated Types:

טיפוסי מניה הם טיפוסים קבועים ברי מניה (כמו למשל קלפים: עלה, תלתן, לב, ויהלום). מבנה הטיפוס:

```
public enum Suit{
SPADES, HEARTS, DIAMONDS, CLUBS }
```

ואז ניתן לבצע על טיפוס ממחלקה זו switch. כמו כן ניתן להוסיף בנאי למחלקה שיקבל כמחרוזת את שם המחלקה, וניתן להוסיף לכל סוג שם, למשל: SPADES("Spades").

טיפוס EnumSet, EnumMap

- כל ערכי הקבוצה באים מ-enum קיים. מאפשר ליצור קבוצה המחזיקה יותר מסוג אחד מה-enum, כלומר יותר מתכונה אחת (כמו צורות היכולות להיות בו"ז: עגולה, מלאה וכו')
- הייצוג הפנימי הוא בביטים (יעיל).
- למשל: Set<someEnumType> s = EnumSet.**of**(someEnumType.type1, <some more types>)
- או: Set<someEnumType> s = EnumSet.**allOf**(someEnumType.class) – יכיל את כל הסוגים.
- EnumMap עובד בצורה דומה.