מעבדה מספר 7

Databases, ORM, JPA, Hibernate

במעבדה זו, נכיר כלים לשמירת מידע לטווח ארוך, וכן טכניקה שתעזור לנו לעבוד מולם בצורה נוחה ומונחית עצמים.

חלק א' – מסדי נתונים (Databases)

עד כה, הכרנו שתי דרכים לשמור מידע בתוכניות שלנו:

- משתנים
- קבצים

לשתי הדרכים הנ״ל יתרונות משלהן: שימוש במשתנים הינו מהיר במיוחד, כיוון שהם נשמרים בזיכרון, ואילו קבצים נשמרים לטווח ארוך. עם זאת, ישנם חסרונות ברורים:

- כיוון שמשתנים נשמרים בזיכרון ה־RAM של המחשב, ברגע שנצא מהתוכנית הם כבר לא ישמרו.
- שימוש בקבצים הינו איטי יחסית, מסורבל ויכול להיות בעייתי כאשר רוצים לאחסן קבצים מחוץ למחשב המקומי.

נציג פתרון שלישי:

מסד נתונים או **בסיס נתונים** (Database) הוא אמצעי המשמש לאחסון מסודר של נתונים במחשב, לשם אחזורם ועיבודם. (ויקיפדיה).

קיימים מספר סוגי מסדי נתונים – אנחנו נתמקד באחד, שבו גם תשתמשו בפרויקט הסופי – מסדי נתונים טבלאיים/רלציוניים (Relational). בסוג זה, הנתונים נשמרים בצורה של **טבלאות**, כאשר כל טבלה מכילה מידע על ישות מסוימת (למשל: סטודנט במערכת ניהול של האוניברסיטה). בכל טבלה יש **רשומות** (שורות), כאשר כל רשומה בטבלה מתייחסת למקרה ספציפי (למשל: סטודנט מסוים). לכל רשומה בטבלה יש **מפתח ראשי**, שמזהה באופן ייחודי את אותה רשומה (בדרך כלל מספר כלשהו, למשל מספור רץ).

למשל, הטבלה הבאה מייצגת נתונים על מספר פריטים בקטלוג: מזהה ייחודי (מספר רץ), מחיר וכמות:

id	price	quantity
1	25	0
2	15	0
3	30	0
4	10	0
5	40	0
6	200	0
7	190	0

לרוב, נידרש לקשר בין רשומות בטבלאות שונות. בתוך מסד הנתונים, ניתן לעשות זאת באמצעות שדה מיוחד בשם **מפתח זר**, אך לא ניכנס לכך – הספרייה שנכיר בהמשך תדאג לקישור בין היישויות השונות עבורנו.

אפשר לומר שמסדי נתונים מהווים פתרון שלישי, שלו מספר יתרונות:

- המידע נשמר בצורה מסודרת וקלה יחסית לשליפה ולעיבוד.
- המידע נשמר לטווח ארוך – מאחורי הקלעים, בדרך כלל המידע נשמר על גבי הדיסק, אך ניהול האחסון נחסך מהמתכנת ומאפשר להקדיש תשומת לב רבה יותר לאספקטים אחרים וחשובים לא פחות.
- ניתן לאחסן את המידע במחשב/שרת ייעודי בצורה פשוטה יחסית.
- מספר תוכניות יכולות לכתוב לאותו מסד נתונים בו־זמנית – נשים לב שלא תמיד נרצה זאת.

עם זאת, ישנם מספר חסרונות:

- נוספה אחריות חדשה למערכות שאנחנו מפתחים – עלינו לדאוג לתקשורת מול מסד הנתונים, על כל המשתמע מכך – למשל, מה קורה כאשר אין תקשורת עם המסד? כיצד נדאג שמשתמש זדוני לא יוכל להשתמש בנתונים שלנו כנגדנו? (למשל, ע״י טכניקה בשם SQL Injection). כיצד נדאג שמסד הנתונים לא יהווה צוואר בקבוק במערכת?
- לרוב, ישנה פחות גמישות – כלומר, כדאי לתכנן מראש כיצד הנתונים יהיו מאוחסנים.
  - בהרצאות, תלמדו כיצד למדל מחלקות באמצעות UML. בין היתר, קיימת חלוקה של מחלקות לשלושה רבדים, כאשר אחד מהם, ה**יישות** (Entity), יכול לייצג את אופן שמירת המידע (על הקישור בין מסד הנתונים למחלקות נלמד בהמשך המעבדה).
  - עוד דרך למדל את שמירת הנתונים במסד נתונים נקראת **דיאגרמת ישויות-קשרים** (Entity Relationship Diagram – ERD), אך לא נעסוק בה בקורס.
- מסדי נתונים רלציוניים שומרים מידע בצורת טבלה.
  - נראה כיצד לעבוד מולם בצורה מונחית עצמים בהמשך.

במעבדה 6 למדנו על שפת SQL (Structured Query Language) המשמשת לעבודה מול מסד הנתונים. תוכלו ללמוד עליה עוד כאן.

אנחנו נשתמש במסד נתונים בשם MySQL. יש מספר דרכים להשתמש בו:

1. התקנה מקומית.
2. הרצה של container באמצעות docker.
3. שימוש בשירות המספק גישה לשרת MySQL מרוחק .
4. התקנה על שרת בענן – לא מעט חברות מספקות גישה בחינם לסטודנטים, למשל: AWS, Azure. מומלץ בחום ☺

לפני שנמשיך, נכיר קונספט חשוב: **תנועה** (נקראת גם טרנזקציה – transaction). במערכות מידע, תנועה היא פעולה לוגית לשינוי נתונים, המורכבת מסדרת פעולות בדידות. לעתים, תנועה מכונה גם **יחידת עיבוד לוגית**, מאחר שכל הפעולות הבדידות המרכיבות אותה חייבות להתבצע כיחידה אחת, או לא להתבצע כלל. זהו עקרון חשוב – כל תנועה היא אטומית.

בהקשר שלנו, תנועה תהיה לרוב פעולה לוגית שמתבצעת מול מסד הנתונים ולא הגיוני לפרקה ליחידות שונות. לדוגמה: יצירת מידע התחלתי עשויה להיחשב כתנועה.

חלק ב׳ – אנוטציות (Annotations) ב־Java

בשפת Java, אנוטציות הינן דרך להוסיף מידע לקוד המקור. ניתן להוסיפן למחלקות, מתודות, משתנים וחבילות. בניגוד להערות, האנוטציות נשמרות לאחר הידור וניתן להשתמש בהן בזמן ריצה.

מבחינת תחביר, אנוטציות מוגדרות ע״י הסימן ״@״ ואחריו מילה מסוימת. למשל, נוכל להוסיף אנוטציה בשם MyAwesomeAnnotation למחלקה Car כך:

```
@MyAwesomeAnnotation
public class Car {
    /* Stuff */
}
```

קיימות מספר אנוטציות מובנות בתוך Java – שלוש מהן מאוד שימושיות:

- @Override –מוודאת שהמתודה המסומנת אכן דורסת (overrides) מתודה אחרת. אם המתודה לא נמצאת באחת מהמחלקות שיורשים, או באחד הממשקים שמממשים, אותה אנוטציה גורמת לשגיאת קומפילציה.
- @Deprecated – מסמנת מתודה כמיושנת – כלומר כאחת שכבר עדיף שלא להשתמש בה בקוד חדש שנכתב. שימוש במתודה עם האנוטציה הנ״ל גורם לאזהרה בשלב הקומפילציה.
- @SuppressWarnings – מורה לקומפיילר שלא להציג אזהרות מסוימות (ניתן להגדירן ע״י פרמטר לאותה אנוטציה).

בנוסף, השימוש באנוטציות נפוץ מאוד בתשתיות תוכנה וספריות שונות – בין היתר בספרייה Hibernate, שאותה נכיר עוד מעט.

חלק ג׳ – מיפוי אובייקטי-רלציוני (Object-Relational Mapping – ORM)

זוהי טכניקה לקישור בין אובייקטים לבין טבלאות בבסיסי נתונים רלציוניים. בעצם, שימוש בה מאפשר לנו ״לדבר״ עם מסד הנתונים בשפה מונחית עצמים – כלומר לשמור מחלקות שבהן שדות, וכן לשלוף מידע בצורה של מופעים של מחלקות, במקום לעבוד ישירות מול טבלאות ועמודות (המיוצגות כמחרוזות).

למשל, אם נרצה לשמור מידע על פריטים בקטלוג, כמו בטבלה מקודם, נוכל לייצג פריט בודד ע״י מחלקה:

```
public class CatalogItem {
    private int id;
    private double price;
    private int stockAmount;

    /* Constructors, getters and setters, as required */
}
```

במצב כזה, נוכל להתייחס לקטלוג כאל רשימה של מופעים של המחלקה. בנוסף, על מנת לשמור פריט חדש, כל שיהיה עלינו לעשות הוא ליצור מופע חדש של המחלקה CatalogItem ולשמור אותו במסד הנתונים.

שימוש ב־ORM, לרוב, מקטין את כמות הקוד שנדרשים לכתוב. יש גם חיסרון בולט – לעתים השימוש עשוי להיות איטי יותר מעבודה ישירות מול מסד הנתונים באמצעות SQL. עם זאת,

לשימושים פשוטים יחסית (כמו בקורס), שאינם דורשים אופטימיזציה, אותה פשרה בביצועים אינה מורגשת.

מטלה 1

ענו על השאלות הבאות:

1. מה התפקיד של ORM?
2. באילו מקרים רצוי להשתמש ב־SQL ולא ב־ORM?
3. תנו דוגמה לשימוש באנוטציה **שאינה** מובנית ב־Java.

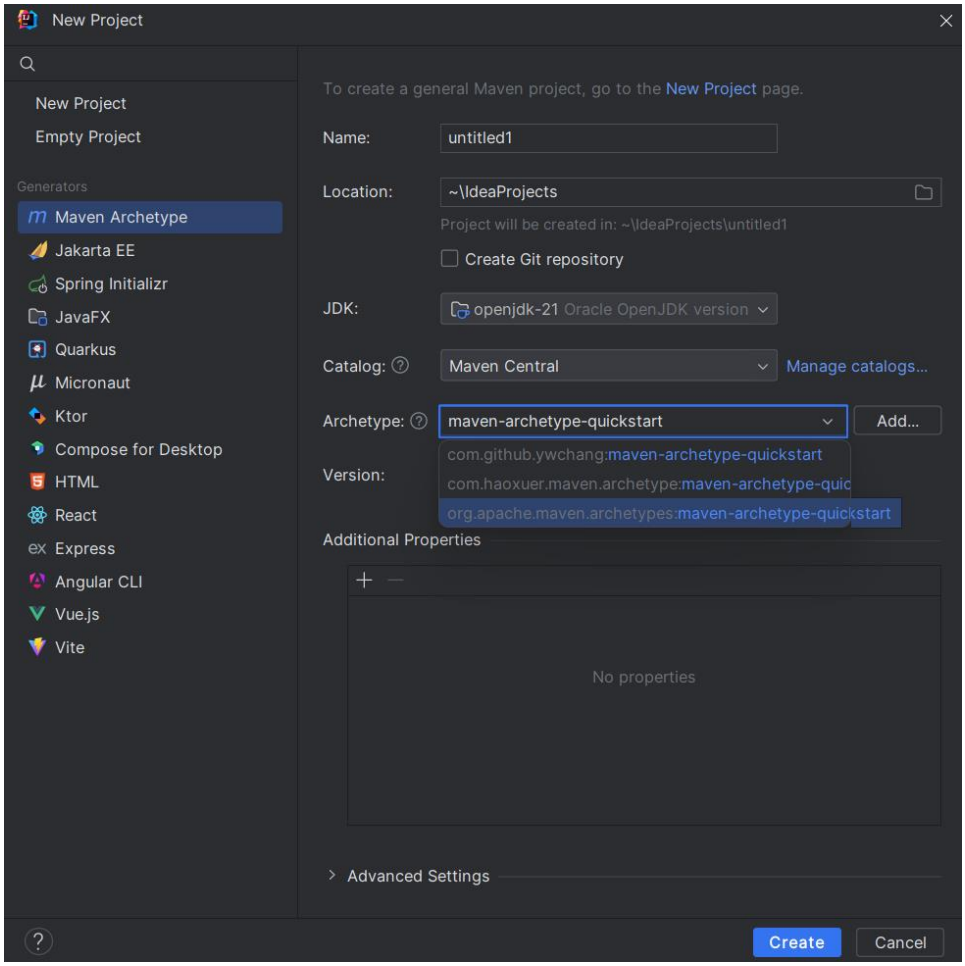
חלק ד' – Java Persistance API, Hibernate

בשפת Java, מוגדר API נוח לעבודה עם נתונים, בשם Java Persistance API (JPA). אותו API מאפשר לנו להגדיר ישויות ושדות, וכן לציין עבורם פרטים נוספים הקשורים לאופן השמירה במסד הנתונים באמצעות אנוטציות.

מימוש נפוץ של JPA מגיע כחלק ספריית ORM בשם Hibernate, שתומכת במספר רב של מסדי נתונים – בהם MySQL, שבו נשתמש.

ניצור פרויקט לדוגמה, ותוך כדי נלמד כיצד להשתמש בספרייה:

1. כמו מקודם, נתחיל עם ה־archetype של quickstart:



במקרה הזה, הדומיין יהיה org.example ושם ה־artifact יהיה hello_hibernate (תוכלי למצוא אלה ב advanced settings). אל תשכחו לשנות את גרסת Java ל־13 ב־pom.xml ולהוסיף את plugin של exec-maven-plugin כמו שעשיתם במעבדה 1.

2. נוסיף שתי תלויות: hibernate-core ו־mysql-connector-java

```
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.19</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.12.Final</version>
    </dependency>
</dependencies>
```

הראשונה היא הספרייה עצמה (שימו לב לקחת את גרסה 5 ולא 6 שעדיין אינה יציבה) והשנייה היא דרייבר שיאפשר להתחבר למסד נתונים MySQL.

לא לשכוח לעדכן את הפרויקט בשלב זה! (קליק ימני על ה־pom.xml ואז Maven- >Update Project)

3. ניצור תיקייה בשם resources ב־src/main
4. בתיקיית ה־resources שיצרנו, ניצור קובץ חדש בשם hibernate.properties, שבו נגדיר כיצד להתחבר למסד הנתונים:

```
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
hibernate.connection.driver_class = com.mysql.cj.jdbc.Driver
hibernate.show-sql = true
# Change if you don't want Hibernate to create the tables
hibernate.hbm2ddl.auto = create
# Change 'localhost' to your server address and 'example' to your database
name
hibernate.connection.url = jdbc:mysql://localhost/example
# Change 'root' to your username
hibernate.connection.username = root
# Change 'password' to your password
hibernate.connection.password = password
```

ההערות להסברים בלבד ויש להסירן בקובץ.

את שלוש השורות הראשונות לא נשנה, ואת שאר השורות נשנה בהתאם להערות. נשמור את הקובץ.

הערה: לפעמים משתמשים במונח schema במקום database (example במקרה הזה).

5. נרצה להוסיף ישות ראשונה: מכונית (לשם הדוגמה). למכונית יש מזהה יחודי, שהוא מספר רץ, לוחית רישוי, מחיר ושנת יצור. ניצור מחלקה חדשה בשם Car במחלקה org.example:

```java
package org.example;

import javax.persistence.*;

@Entity
@Table(name = "cars")
public class Car {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String licensePlate;

    private double price;

    @Column(name = "manufacturing_year")
    private int year;
    public Car() { }
    public Car(String licensePlate, double price, int year) {
        super();
        this.licensePlate = licensePlate;
        this.price = price;
        this.year = year;
    }

    public String getLicensePlate() {
        return licensePlate;
    }

    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getYear() {
        return year;
    }
```

```
    public void setYear(int year) {
        this.year = year;
    }

    public int getId() {
        return id;
    }
}
```

נשים לב למספר דברים:

- מומלץ לתת גם שמות לטבלאות בעזרת האנוטציה @Table.
- **עליכם לדאוג שאתם לא משתמשים בשמות שהם מילים שמורות. למשל, YEAR הינה מילה שמורה של MySQL ולכן ניתן לעמודה שלה שם אחר.** תוכלו למצוא רשימה של המילים השמורות כאן: https://dev.mysql.com/doc/refman/8.0/en/keywords.html.
- ביישויות אין חובה שתהיינה מתודות פרט לבנאי, getters ו־setters. ניתן להוסיף מתודות כרצונכם.
- כיוון שהמזהה הוא מספר רץ, אנחנו לא מקבלים אותו כפרמטר בבנאי.
- כל מחלקה של ישות צריכה לקבל את האנוטציה Entity.
- מזהה (Id) מקבל אנוטציה של Id. אם אנחנו גם רוצים שיהיה בסדר רץ, עלינו להוסיף את האנוטציה של GeneratedValue עם הפרמטר של strategy. במקרה שלנו (ולרוב) מספיק להשתמש ב־GenerationType.IDENTITY בתור Strategy, אך ישנן עוד מספר אפשרויות.

6. נשנה את המחלקה App כך:

```
package org.example;

import java.util.List;
import java.util.Random;

import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

public class App
{

    private static Session session;

    private static SessionFactory getSessionFactory() throws
HibernateException {
```

```
        Configuration configuration = new Configuration();

        // Add ALL of your entities here. You can also try adding a whole
package.
        configuration.addAnnotatedClass(Car.class);

        ServiceRegistry serviceRegistry = new
StandardServiceRegistryBuilder()
                .applySettings(configuration.getProperties())
                .build();

        return configuration.buildSessionFactory(serviceRegistry);
    }

    private static void generateCars() throws Exception {
        Random random = new Random();

        for (int i = 0; i < 10; i++) {
            Car car = new Car("MOO-" + random.nextInt(), 100000, 2000 +
random.nextInt(19));
            session.save(car);
            /*
             * The call to session.flush() updates the DB immediately
without ending the transaction.
             * Recommended to do after an arbitrary unit of work.
             * MANDATORY to do if you are saving a large amount of data -
otherwise you may get cache errors.
            */
            session.flush();
        }
    }

    private static List<Car> getAllCars() throws Exception {
        CriteriaBuilder builder = session.getCriteriaBuilder();
        CriteriaQuery<Car> query = builder.createQuery(Car.class);
        query.from(Car.class);
        List<Car> data = session.createQuery(query).getResultList();
        return data;
    }

    private static void printAllCars() throws Exception {
        List<Car> cars = getAllCars();
        for (Car car : cars) {
            System.out.print("Id: ");
```

```
            System.out.print(car.getId());
            System.out.print(", License plate: ");
            System.out.print(car.getLicensePlate());
            System.out.print(", Price: ");
            System.out.print(car.getPrice());
            System.out.print(", Year: ");
            System.out.print(car.getYear());
            System.out.print('\n');
        }
    }

    public static void main( String[] args ) {
        try {
        SessionFactory sessionFactory = getSessionFactory();
        session = sessionFactory.openSession();
        session.beginTransaction();


        generateCars();

        printAllCars();

        session.getTransaction().commit(); // Save everything.

        } catch (Exception exception) {
            if (session != null) {
                session.getTransaction().rollback();
            }
            System.err.println("An error occured, changes have been
rolled back.");
            exception.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

בעצם, אנחנו עושים את הדברים הבאים:

- יוצרים עשרה עצמים חדשים מטיפוס Car.
- שומרים אותם במסד הנתונים.
- מדפיסים את האובייקטים שכרגע קיימים במסד הנתונים.

מספר דגשים חשובים:

- שימו לב להערה במתודה getSessionFactory,– עליכם להגדיר את **כל** המחלקות שמייצגות ישויות באופן דומה למחלקה Car.

- במקרה הזה, הגדרנו את שמירת המידע והדפסתו כתנועה אחת. ב־Hibernate, תנועה מתחילה בעזרת המתודה beginTransaction ומסתיימת בעזרת המתודה commit (אם התנועה הושלמה בהצלחה ונרצה לשמור את התוצאות שלה) או rollback (אם התנועה לא הצליחה ונרצה לבטלה).
- במקרה של חריגה, אנחנו מבטלים את השינויים שעשינו לפני כן כחלק מאותה תנועה. זו בחירה שלכם, אך לרוב מומלץ לעשות זאת.
- **חובה** לסגור את ה־session (בעצם – החיבור למסד הנתונים) בסיום התכנית. אחרת – יכולות לצוץ לא מעט בעיות.
- **חובה** לבצע commit ל־session לאחר שסיימנו לבצע שינויים, על מנת שהם יתעדכנו במסד הנתונים.
- **רצוי מאוד** לבצע flush לאחר כל ״יחידה״ (שרירותית לחלוטין ונתונה לשיקולכם, למשל – אחרי הוספה של כל הישויות ההתחלתיות מטיפוס מסוים, או אחרי הוספה של ישות בודדת, אם היא מכילה כמות רבה של פריטים).
- ב־Console באקליפס יתכן שתראו שורות צבועות באדום. אלה **אינן** בהכרח שגיאות. שימו לב לכיתוב בהתחלה – אם נכתב INFO אז מדובר במידע ש־hibernate כותבת – ניתן לשלוט בכך, אך מומלץ להשאיר את כל הלוגים למטרות debugging, לפחות בתור התחלה.
- מומלץ לבנות גרסה גנרית של printAllCars ולהשתמש בה. אפילו טוב יותר - לשים אותה במחלקה סטטית, כך שתוכלו להעבירה בין פרויקטים ולהשתמש בה שוב ושוב. אם נתקעתם – יש בדוגמה במודל מתודה גנרית.
- קיים תיעוד רב באינטרנט על Hibernate – רצוי (ואף חובה – החומר הנלמד כאן רחוק מלכסות את כל הפונקציונליות שעליכם לדעת) לחפש על כך באינטרנט.

מטלה 2

ענו על השאלות הבאות (זכרו – Google is your friend):

4. מהו קשר יחיד ליחיד? תנו דוגמה לשימוש בו.
5. מהו קשר יחיד לרבים? תנו דוגמה לשימוש בו.
6. מהו קשר רבים לרבים? איך הוא ממומש? תנו דוגמה לשימוש בו.

מטלה 3

יש לפתור את תרגיל מעבדה 7 המפורט בקובץ Lab7-assignment.

הוראות הגשה לדוח מעבדה מס׳ 7:

1. ההגשה בזוגות בלבד באמצעות הגשה אלקטרונית. ניתן להגיש מספר פעמים עד לשעת הסיום.
2. יש להגיש קובץ zip אחד. שם קובץ ה־zip הוא מספרי תעודות הזהות של הסטודנטים מופרדים בגרש תחתון. ה־zip כולל שני קבצים:
   a. קובץ PDF ובו תשובה למטלות 1 ו־2 כמפורט להלן.
   b. קובץ JAR בר הרצה של תרגיל המעבדה, הכולל בתוכו גם את קובצי המקור.
3. אנא הקפידו על הוראות ההגשה.

ע ב ו ד ה נ ע י מ ה!