01 JDBC

* JDBC is a standard Java API for database-independent   
  connectivity
* Includes APIs for:
  + Making a connection to a database
  + Creating and executing **SQL** queries in the database
* Viewing & Modifying the resulting records
* JDBC **API** – provides the connection between the application and the driver manager
* JDBC **Driver Manager** – establishes the connection with the   
  correct driver
  + Supports multiple drivers connected to different types of   
    databases
* JDBC **Driver** - handles the communications with the database
* JDBC **API** provides several interfaces and classes:
  + **DriverManager** – matches requests from the application with the proper DB driver
  + **Driver** – handles the communication with the DB server
  + **Connection** – all methods for contacting a database
  + **Statement** – methods and properties that enable you to send SQL
  + **ResultSet** – retrieved data (set of table rows)
  + **SQLException**
* **ResultSet** maintains a **cursor** pointing to its **current row of data**
  + Not updatable
  + Iterable only once and only from the first row to the last row
  + Provides getter methods for retrieving column values from the current row
  + Retrieved information is reached by getter methods:
  + E.g.:
  + getString("column\_name")
  + getDouble("column\_name")
  + getBoolean("column\_name") etc.
  + The driver converts the underlying data to the Java type
  + The java.sql package provides all previously mentioned JDBC   
    classes
  + In order to work with JDBC we need to download a MySQL Driver – Connector/J
  + Connection with the database is established via **connection**   
    **string**
  + jdbc:<driver protocol>:<connection details> - jdbc:mysql//localhost:3306/my\_db
  + E.g. connection from previous demo:

**Connection c = DriverManager.getConnection("jdbc:mysql://localhost:3306/soft\_uni", props);**

* The JDBC **Statement interface** defines the methods and properties that enable you to send SQL commands to the database.

|  |  |
| --- | --- |
| **Interfaces** | **Recommended use** |
| Statement | For general-purpose access to your database and static SQL statements at runtime. Cannot accept parameters. |
| PreparedStatement | For SQL statements used many times. Accepts parameters. |
| CallableStatement | Used for stored procedures. Accepts parameters. |

**What is ORM?**

* **Technique** for **converting data** betweenincompatible type systems using **object-oriented programming** languages
* **Object-Relational Mapping** (ORM) allows   
  manipulating databases **using common classes and   
  objects**
  + **Java/C#/etc. classes** 🡺 **Database Tables**
  + **Database Tables** 🡺 **Java/C#/etc. classes**

**Why do we need ORM?**

* In OOP, data-management tasks act on **objects** that are almost always **non-scalar** values
* Many **database** can only store and manipulate **scalar** values, organized within **tables**
* We must **manually** convert values into groups of simpler values to store in DB and convert them back when we retrieve data

**JDBC vs ORM**

* The main difference, between JDBC and ORM, is **complexity**
* **JDBC/SQL** - If the application is simple as to present data directly from the database
* **ORM** - If the application is domain driven and the relations among objects is complex

**ORM Frameworks: Features**

* **ORM frameworks** typically **provide** the following functionality:
  + **Automatically generate SQL** to perform data operations as persist, update, delete, merge, createQuery and so on.
  + **Object model from database schema** (DB First model)
  + **Database schema from object model** (Code First model)
  + We can use and specific ORM Query Language as **HQL** or **SQL**

**ORM Advantages**

* **Productivity:** Eliminates repetitive code, Generates database automatically
* **Maintainability:** Fewer lines of code, Easier to manage object model changes
* **Performance:** Lazy loading, Caching
* **Database vendor independence** The database is abstracted, Can be configured outside the application

**ORM Disadvantages**

* **Reduced performance:** Due to overhead or auto generated SQL
* **Reduces flexibility:** Some operations are hard to implement
* **Lose understanding:**  What the code is actually doing - the developer is more in control using SQL

**Hibernate Framework**

* Hibernate is a Java ORM framework
  + Mapping an object-oriented model to a relational database
    - It is implemented by the configuration of an **XML** **file** or by using **Java** **Annotations**
  + Maintain the database schema
* Different approaches to **Java** **ORM**:
  + POJO (Plain Old Java Objects) + XML mappings
    - A bit old-fashioned, but very powerful
    - Implemented in the "classical" Hibernate
  + Annotated Java classes (**POJO**) mapped to DB tables
    - Based on Java annotations and XML
    - Easier to implement and maintain
  + Code generation – tools

**About JPA**

* What is Java Persistence API (JPA)?
  + Database persistence technology   
     for Java (**official** **standard**)
    - Object-relational mapping (ORM) technology
    - Operates with POJO entities with annotations or XML mappings
    - Implemented by many **ORM** **engines**:   
       **Hibernate**, **EclipseLink**, etc.
* **Annotations**
* **@Entity** - Declares the class as an entity or a table
* **@Table** - Declares table name
* **@Basic** - Specifies non-constraint fields explicitly
* **@Transient** - Specifies the property that is not persistent, i.e., the value is never stored in the database
* **@Id** - Specifies the property, use for identity (primary key of a table) of the class
* **@GeneratedValue** - specifies how the identity attribute can be initialized
  + Automatic, manual, or value taken from a sequence table
* **@Column** -Specifies the column attribute for the persistence property

**JPA Write Data Methods**

* **persist()** – persists given entity object into the DB (SQL INSERT)
* **remove()** – deletes given entity into the DB (SQL DELETE by primary key)
* **refresh()** – reloads given entity from the DB (SQL SELECT by primary key)
* **detach()** – removes the object from the persistence context(PC)
* **merge()** – synchronize the state of detached entity with the PC
* **contains()** - determine if given entity is managed by the PC
* **flush()** – writes the changes from PC in the database

**JPA Read Data Methods**

* **find()** - execute a simple Select query by primary key

**JPA Merge Objects**

* Merges the state of **detached** entity into a **managed** **copy** of the detached entity.
  + Returned entity has a different Java identity than the detached one

public Student storeUpdatedStudent(Student student) {

return entityManager.merge(student);

}

* May invoke SQL SELECT

**Java Persistence API Inheritance**

* Inheritance is a fundamental concept in most programming languages
  + SQL does not support this kind of relationships
* Implemented by any JPA framework by **inheriting** and **mapping** **Entities**

**JPA Inheritance Strategies**

* Implemented by the **javax.persistence.Inheritance** annotation
* The following mapping strategies are used to map the entity data to the underlying database:
  + A single **table per class** hierarchy
  + A table per **concrete entity class**
  + "**Join**" strategy – mapping common fields in a single table

**Table Per Class Strategy**

* **Table creation for each entity**
  + A table defined for each concrete class in the inheritance
  + Allows inheritance to be used in the object model, when it does not exist in the data model
* Querying root or branch classes can be very difficult and **inefficient**
* **Disadvantages**:
  + Repeating information in each table
  + Changes in super class involves changes in all subclass tables
  + No foreign keys involved (unrelated tables)
* **Advantages**:
  + No NULL values – no unneeded fields
  + Simple style to implement inheritance mapping

**Table Per Class: Joined**

* Table is defined for each class in the inheritance hierarchy
  + Storing of that class **only the local attributes**
  + Each table must store object's **primary key**
* **Disadvantages**:
  + Multiple JOINS - for deep hierarchies it may give poor performance
* **Advantages**:
  + No NULL values
  + No repeating information
  + Foreign keys involved
  + Reduced changes in schema on superclass changes

**Table Per Class: Single Table**

* **Simplest** and typically the best performing and best solution
  + A single table is used to store all the instances   
    of the **entire** **inheritance hierarchy**
  + A column for every attribute of every class
  + A **discriminator column** is used to determine to which   
    class the particular row belongs to

**Database Relationships**

* There are several types of database relationships:
  + - **One to One** Relationships
    - **One to Many** and **Many to One** Relationships
    - **Many to Many** Relationships
    - **Self Referencing** Relationships

**Lazy Loading – Fetch Types**

* Fetching – retrieve objects from the database
  + Fetched entities are stored in the **Persistence Context** as cache
* Retrieval of an entity object might cause automatic retrieval of **additional** entity objects
* Fetching Strategies
  + EAGER – retrieves all entity objects reachable through fetched   
    entity
    - Can cause **slowdown** when used with a big data source
  + **LAZY** – retrieves all reachable entity objects **only when fetched**   
    **entity's getter method is called**

**Cascading**

* JPA translates **entity state transitions** to database **DML**   
  statements
  + This behavior is configured through the **CascadeType** mappings
* **CascadeType.PERSIST**: means that save() or persist()   
  operations cascade to related entities
* **CascadeType.MERGE**: means that related entities are merged   
  into managed state when the owning entity is merged
* **CascadeType.REFRESH**: does the same thing for the refresh()   
  operation
* **CascadeType.REMOVE**: removes all related entities association with this setting when the owning entity is deleted
* **CascadeType.DETACH**: detaches all related entities if a "manual detach" occurs
* **CascadeType.ALL**: is shorthand for all of the above cascade operations