Lecture 7

Persistence and Database Motivation

Persistence Defined

- **Persistence** is the ability to store data so that it survives the lifespan of the application.
- If data is only stored in a static field of an application class, it will be lost when the application restarts; a database is needed to *persist* this data.

Database Systems

- A **Database** is a structured collection of data that can be accessed, managed, and updated.
- Databases facilitate persistence.
- **Application Architectures using DBs:**

  ◦ In a standard web application, clients communicate via HTTP, and the web app uses JDBC (Java Database Connectivity) to interact with the database.

  ◦ **Spring Cloud** applications can scale into many app instances that use **one common DB**.

  ◦ **Spring Microservices** (like an Online Shopping Service broken into Account, Product, Cart, and Order servers) typically use **many apps with their own DBs**. This architecture offers independent scaling, flexibility, resilience (failure in one service doesn't impact others), and faster development.

Types of Databases

The two most important categories are Relational/SQL and NoSQL.

Database Type	Description	Key Features	Use Cases

Relational/SQL	Data is represented as relations in tables. It was invented in 1970 and is the most prevalent model.	Uses Structured Query Language (SQL) for complex queries. Maintains strict data integrity and requires structured schemas with tables and relations. Supports ACID compliance.	Financial applications, enterprise systems.
NoSQL	Designed for unstructured or semi-structured data. Does not require fixed schema definitions, offering flexibility.	Handles large amounts of data with high-speed processing.	Social networks, real-time analytics, scenarios prioritizing scalability.

Types of NoSQL Databases:

- **Document Databases:** Store data in JSON-like documents (e.g., MongoDB).
- **Key-Value Stores:** Store data as collections of key-value pairs, providing fast access (e.g., Redis).
- **Column-Family Stores:** Organize data into columns and rows with a flexible schema (e.g., Cassandra).
- **Graph Databases:** Optimized for complex, interconnected data using nodes, edges, and properties (e.g., Neo4j).

Structured Query Language (SQL)

- SQL is a declarative language, composed of three parts:

  1. **DDL (Data Definition Language):** Used for creating the database and defining/controlling the structure of data (e.g., `CREATE TABLE`, `DROP TABLE`).

  2. **DML (Data Manipulation Language):** Used to access and update data (e.g., `INSERT INTO`).

  3. **DCL (Data Control Language):** Used for administering the database (e.g., `GRANT`, `DENY`, `REVOKE`).

Class Diagrams (UML)

- **Class Diagrams** are a modeling language within the Unified Modeling Language (UML). They are the most common UML tool to formalize Object-Oriented (OO) systems and data models.
  - A **Class** is a blueprint.
  - An **Object** is an individual instance or realization of a class blueprint, holding its own copy of the attributes.

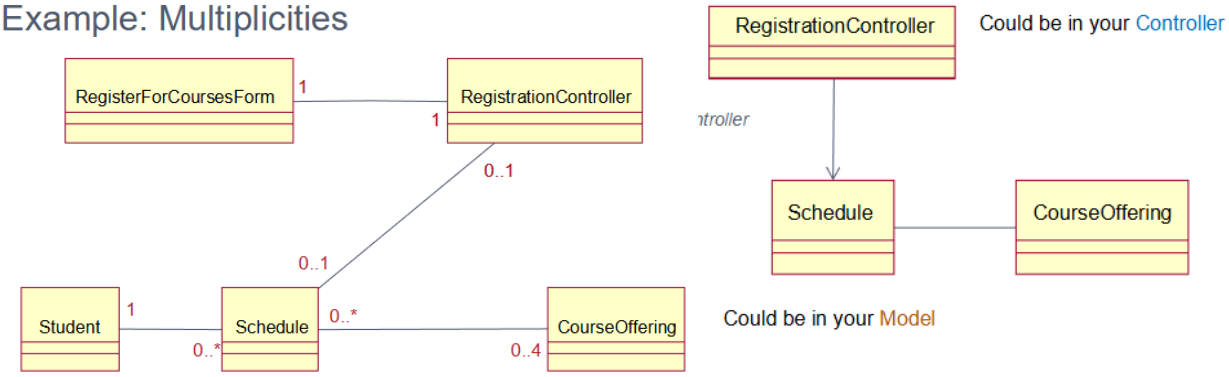
Component	Description	Notation Details
Class Name	The name of the blueprint. A class is a blueprint for objects.	(The class name is typically placed in the top compartment of the rectangle).
Attributes (Fields)	Represents the **data held by the class**.	The standard notation is `[visibility] [/] name [: type] [multiplicity] [=value] [{property}]`. **Visibility** specifies accessibility: **+ public**, **# protected**, **- private**, and **~ package**. **Static attributes** are indicated by being **underlined**.
Methods (Operations)	Represents the **actions the class can perform**.	The standard notation is `[visibility] name ([parameter-list]) [: type] [{property}]`. Visibility rules are the same as for attributes. The formal parameter list may include direction, name, type, multiplicity, value, and property. **Static operations** are **underlined**

Relationships (Associations) Associations are semantic structural relationships between two or more classes.

Relationship Type	Description	UML Notation Tip
Multiplicity (Cardinality)	The number of instances of one class related to **one** instance of another class.	Examples include `1` (Exactly one), `0..1` (Zero or One/Optional), `1..*` (One or More), and `0..*` (Zero or More).
Navigability	Indicates that it is possible to navigate from the associating class to the target class using the association.	A line with an open arrow pointing to the target class.
Generalization (Inheritance)	An "is a kind of" relationship where a subclass inherits	An unfilled triangle pointing to the superclass.

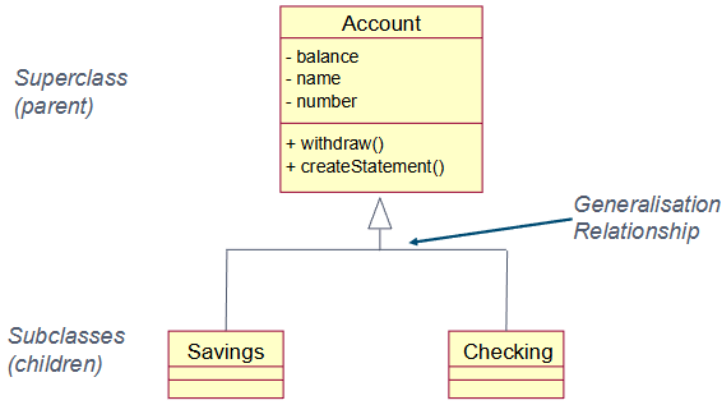
	structure/behavior from a superclass.	
Aggregation	A "part of" relationship where the parts (Class2) and the whole (Class1) have **separate lifetimes**.	A solid line with an **unfilled diamond** on the whole side (Class1).
Composition	A "part of" relationship where the parts (Class2) **live and die with the whole** (Class1).	A solid line with a **filled diamond** on the whole side (Class1).
Dependency	Changes to the definition of one class (Class2) may cause changes to the other (Class1), but not necessarily vice-versa.	A dashed line with an open arrow.

Example: Multiplicities



Example: Single Inheritance

- One class inherits from another.



Java Persistence API (JPA)

JPA is a specification that manages relational data in Java applications. It simplifies database interactions by using **Object-Relational Mapping (ORM)**, which translates Java objects into database structures.

1. Conceptual Mapping JPA bridges the gap between Object-Oriented Java classes and Relational Database structures:

Class Diagram Concept	Java Concept	Relational Database Concept
Class	Class	Table
Attribute/Field	Attribute/Field	Column
Object/Instance	Object/Instance	Tuple (Row)
Association	Attribute/Field (Reference)	Foreign Keys or Intermediate Tables

2. Key JPA Annotations Annotations are used to define how Java objects map to database tables.

- `@Entity`: Marks a class as a database entity, meaning JPA maps this class to a database table. Without it, JPA ignores the class for persistence.
- `@Id`: Designates a field as the unique identifier (primary key) for the entity in the database.

Handling Association Multiplicities in JPA

- JPA uses relationship annotations to manage how Java objects relate to each other, automatically handling schema mapping and complex SQL joins.

Relationship Type	Database Representation	JPA Annotation	Notes
One-to-One	Key on either side of the relationship.	`@OneToOne`	Used, for instance, if a `Hotel` has one `Address`.

One-to-Many	**Foreign key** column in the **Room (many-side)** table referencing the **Hotel (one-side)** table.	`@OneToMany`	A `Hotel` has a `List<Room>`. `@JoinColumn` is often used to join the tables.
Many-to-One	**Foreign key** column on the **Hotel (many-side)** table referencing the **Owner (one-side)**.	`@ManyToOne`	This is the inverse perspective of One-to-Many.
Many-to-Many	Requires an **intermediate join table** with two foreign keys, one referencing each entity.	`@ManyToMany`	One side must be the **owning side** (responsible for managing the join table). The inverse side uses `mappedBy="field_name"` to reference the owning side's field, preventing JPA from creating two join tables.

Analogy for JPA

Think of JPA as a universal translator in the world of data. Java speaks "objects" (classes, fields, methods), and databases speak "tables" (columns, rows, SQL). JPA's annotations are like special grammar rules that let the translator automatically convert your Java object thoughts directly into database language and back again, so you rarely have to write the complex SQL sentences yourself.