Business Informatics Group

Vienna University of Technology

Web Engineering

Introduction to JPA and Hibernate

Philipp Liegl

Business Informatics Group
Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at

Outline of today's talk

- JDBC

- JPA/Hibernate
    - Relationships
    - Persistence Context/Persistence Unit
    - Entity Manager
    - JPQL
    - Hibernate Criteria API

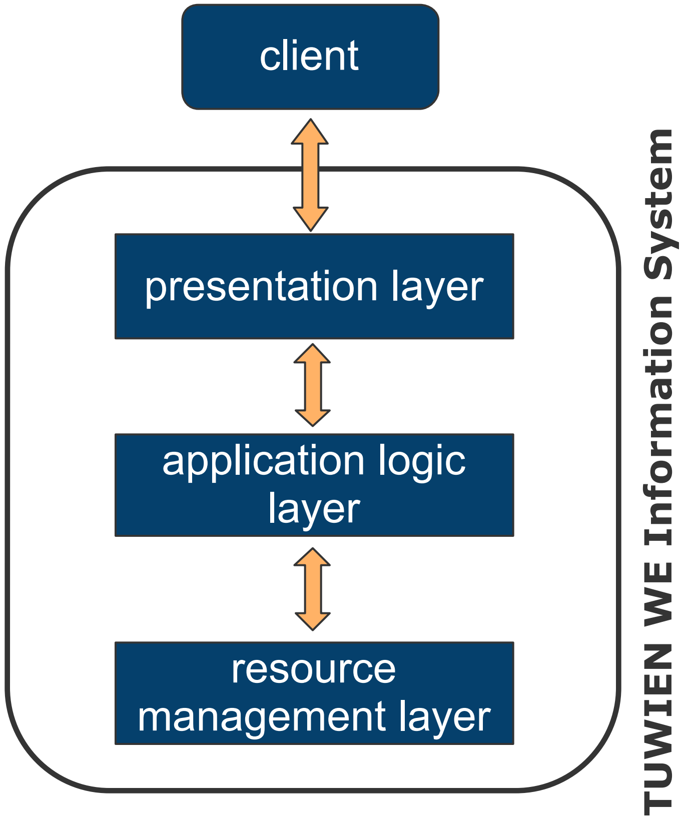
Accompanying examples

https://github.com/pliegl/we2014/tree/master/jpa-sample

Motivation

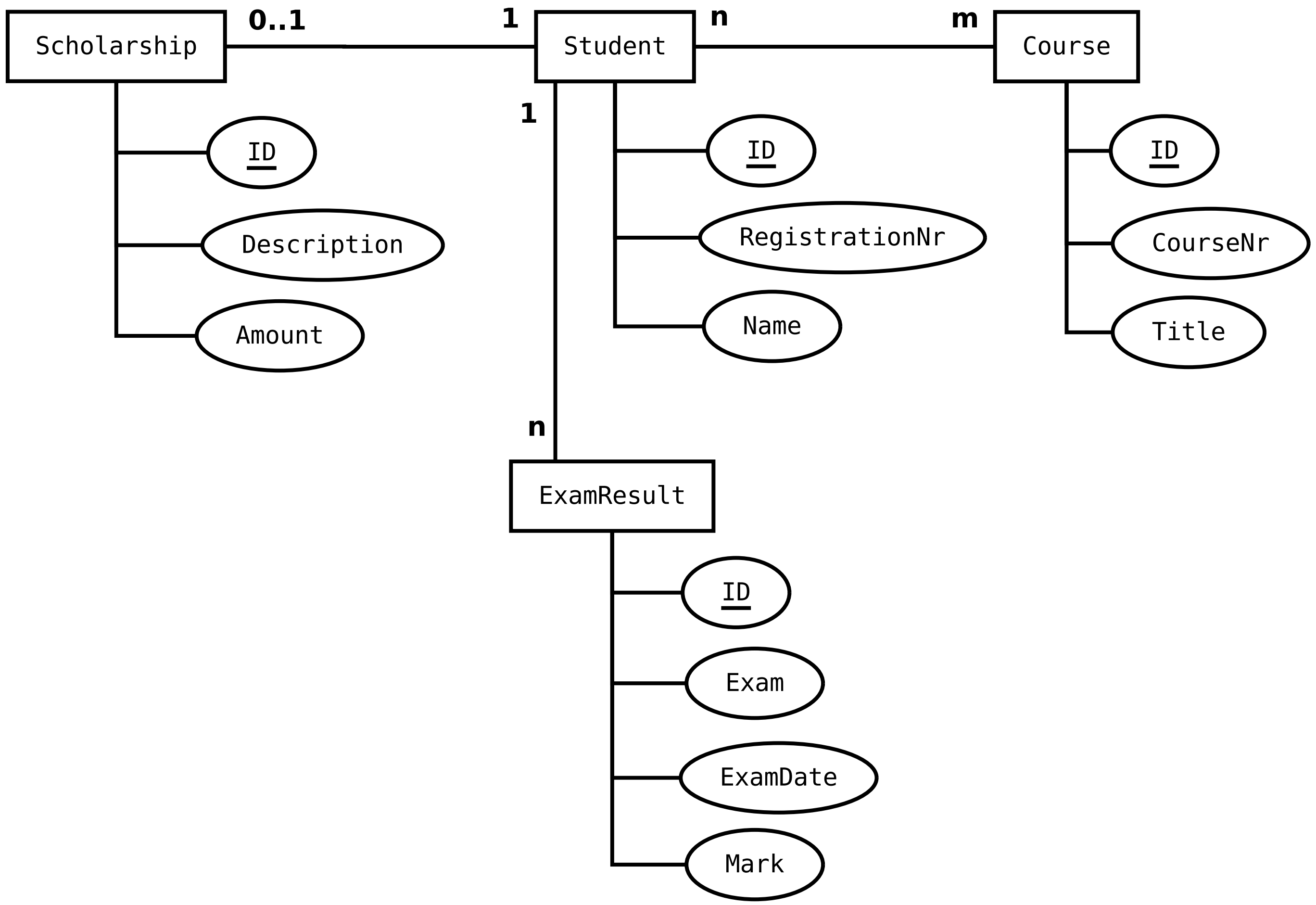
N-Tier Architectures

- Layers of an information system
  - Presentation layer
    - Communication interface to external entities
    - “View” in the model-view-controller
  - Application logic layer (service layer)
    - Implements operations requested by clients through the presentation layer
    - Represents the “business logic”
  - Resource management layer (persistence layer)
    - Deals with different data sources of an information system
    - Responsible for storing and retrieving data



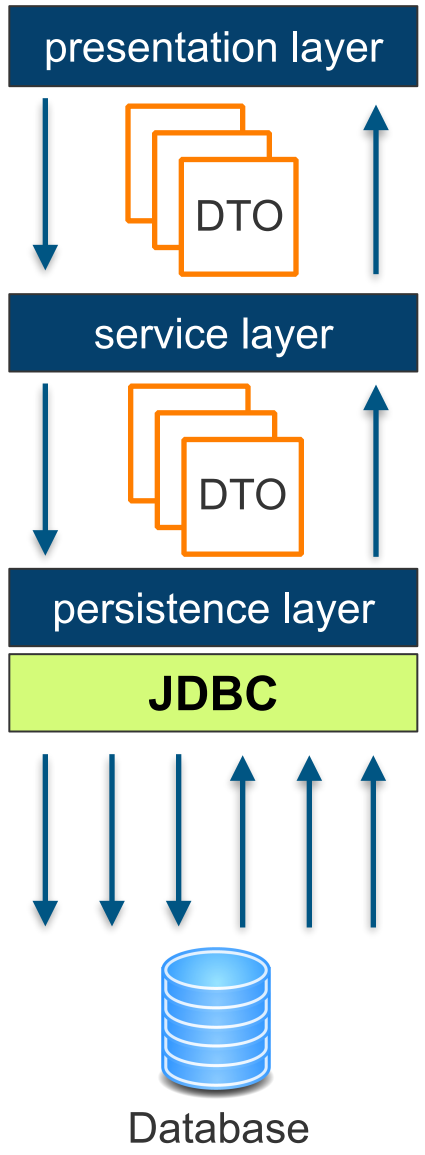
Motivation

Accompanying model



Motivation

Traditional persistence with JDBC



Motivation

JDBC - Java Database Connectivity

- Used to access relational databases from Java programs
- First version released 1996
- Ability to
  - Establish a connection to a database
  - Execute an SQL statement and return results
  - Create parameterized queries
  - Manage database transactions
- Basic Steps
  - Load driver or obtain an already defined data source
  - Establish connection using a JDBC URL
  - Create an **SQL statement** and execute SQL statement
  - If present, process results present in **result sets**
  - Close database resources
  - Commit or rollback transaction, if necessary

JDBC

Insert an entry

```
Connection conn = null;
PreparedStatement stmt = null;

try {
		conn = connection();
		stmt = conn.prepareStatement( "INSERT INTO student VALUES(?, ?, ?)" );
		stmt.setInt( 1, student.getId() );
		stmt.setString( 2, student.getMatrNr() );
		stmt.setString( 3, student.getName() );
		stmt.executeUpdate();
		stmt.close();
} catch (Exception e) {
		e.printStackTrace();
} finally {
		if (stmt != null) {
		   stmt.close();
		}
		if (conn != null) {
		   conn.close();
		}
}
```

JDBC

Retrieve an entry

```
Connection conn = null;
PreparedStatement stmt = null;
ResultSet rs = null;
try {
		conn = connection();
		stmt = conn.prepareStatement( "SELECT id, matnr, name FROM student
						 WHERE id=?" );
		stmt.setInt( 1, id );
		rs = stmt.executeQuery();
		rs.next();

		Student student = new Student();
		student.setId( rs.getInt( 1 ) );
		student.setMatrNr( rs.getString( 2 ) );
		student.setName( rs.getString( 3 ) );

		rs.close();
		stmt.close();
		return student;
} catch (Exception e) {
		e.printStackTrace();
} finally {
		…
}
```

Java Persistence API (JPA)

Introduction

- Specification for the management of persistence and object/relational mapping with Java
  - Persistence: Data objects shall outlive the JVM app

- Objective: provide an object/relational mapping facility for Java developers using a Java domain model and a relational database
  - Map Java POJOs to relational databases (which are one type of persistence)

- Standardized under the Java Community Process Program with contributions from Hibernate, TopLink, JDO, and the EJB community

- Hibernate: JPA implementation with additional “native” features, e.g.,
  - HQL (Hibernate Query Language) - similar to JPQL, but with some extensions
  - Criteria API

Object Relational Mapping

Reasons for using ORM

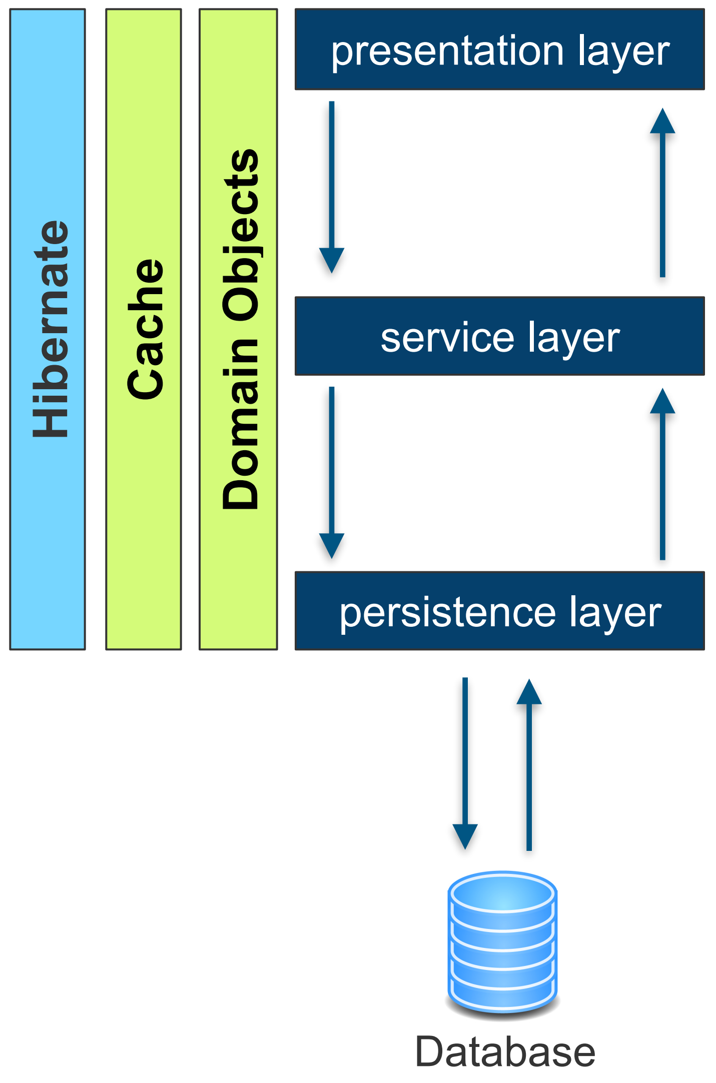
- In an application we want to focus on **business concepts**, not on the relational database structure
- Abstract from the "by-hand" communication with the DB (e.g., via JDBC)
- Allow for an automatic synchronization between Java Objects and the underlying database
- Portability
  - Mostly DB independent (with the exception of some types of features, such as identifier generation)
  - Query abstractions using e.g. JPQL - the vendor specific SQL is auto-generated
- Performance
  - Object and query caching is automatically done by the ORM

Persistent Entities

Basics

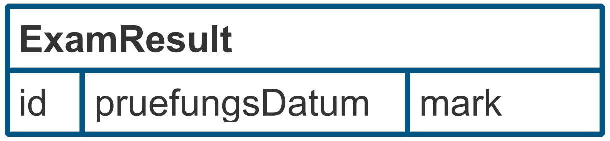
- Are POJOs (Plain Old Java Objects)
- Lightweight persistent domain object
- Typically represent a table in a relational database
- Each entity instance corresponds to one row in that table
- Have a persistent identity
- May have both, persistent and transient (non-persistent) state
  - Simple types (primitive data types, wrappers, enums)
  - Composite types (e.g., Address)
  - Non-persistent state (using identifier `transient` or `@Transient` annotation)

Persistence with Hibernate



Simple Mapping

Enhance Java domain classes with JPA annotations



```
@Entity
public class ExamResult {

    @Id
    private Long id;

    @Column(name = "prufungsDatum")
    @Temporal(TemporalType.DATE)
    private Date examDate;

    private int mark;

    @Transient
    private String examLocation;


    //Getter and setters omitted

}
```

Important annotations	
@Entity	Specifies that the class is an entity
@Id	Specifies the primary key of an entity
@Temporal	Must be specified for fields of type java.util.Date and java.util.Calendar
@TemporalType	Type used to indicate a specific mapping of java.util.Date or java.util.Calendar. Allowed values: - DATE - TIME - TIMESTAMP
@Transient	Specifies that the field is not persistent

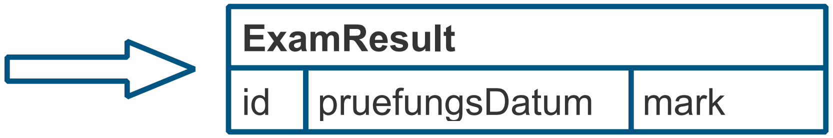
Simple Mapping

Inheritance

```
@MappedSuperclass
public class BaseEntity {

    @Id
    @GeneratedValue
    protected Long id;

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

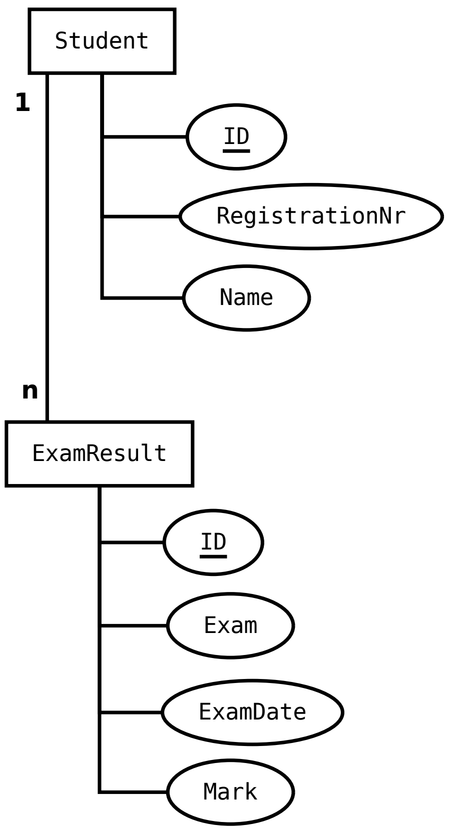
Important annotations	
@MappedSuperclass	Designates a class whose mapping information is applied to the entities that inherit from it. A mapped superclass has no separate table defined for it.
@GeneratedValue	The `GeneratedValue` annotation may be applied to a primary key property or field of an entity or mapped superclass in conjunction with the `Id` annotation.

```
@Entity
public class ExamResult extends BaseEntity {

    @Column(name = "prufungsDatum")
    @Temporal(TemporalType.DATE)
    private Date examDate;

    private int mark;

    @Transient
    private String examLocation;

    //Getter and setters omitted

}
```



ExamResult		
id	pruefungsDatum	mark

Object-oriented vs. SQL



OO: **Student** *owns* the **ExamResults**
Usually: no ExamResult without a student

```
public class Student extends BaseEntity {

    private String registrationNumber;
    private String name;
    private List<ExamResult> examResults;
…
}
```

Does not exist in the DB, but is simulated using an SQL query

```
public class ExamResult extends BaseEntity {

    private Date examDate;
    private String exam;
    private int mark;
    private Student student;
…
}
```

SQL:
- ExamResult contains a foreign key to the Student it belongs to
- The ExamResult owns (contains) the connection
- This is opposite to the OO perspective

Entity relationships

One-to-one, **one-to-many**, **many-to-many**, **many-to-one** relationships among entities

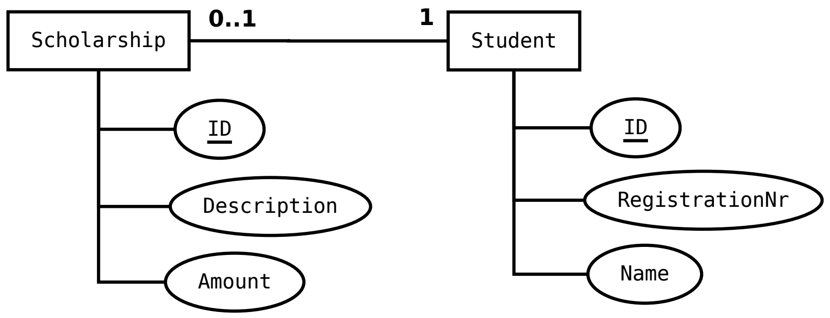
- bi-directional or uni-directional
- Support for different Collection types, e.g., List, Set, Map, etc.

Need to specify the **owning** side in relationships

- Owning side table has the foreign key
- OneToOne relationship - the side where the foreign key is specified
- OneToMany, ManyToOne - the “many” side

Relationship mapping

Example using a unidirectional mapping

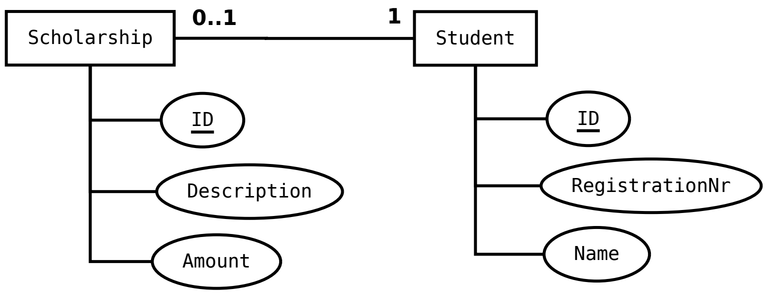


Four different options:

1. Using an embedded table, where Scholarship is the embedded table. (see example)
2. Scholarship and Student are separate tables. The primary key of Scholarship has a foreign key constraint on the primary key of the "owning" Student (using @PrimaryKeyJoinColumn annotation)
3. Scholarship and Student are separate tables. Student holds a foreign key which references the primary key of Scholarship. The foreign key has a unique constraint.
4. Scholarship and Student are separate tables. Scholarship holds a foreign key which references the primary key of Student. The foreign key has a unique constraint. (see example)

Relationship mapping

Unidirectional OneToOne using an embedded table



Important annotations	
@Embedded	Defines a persistent field or property of an entity whose value is an instance of an embeddable class. The embeddable class must be annotated as `Embeddable`.
@Embeddable	Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

```
@Entity
public class EmbeddedStudent extends BaseEntity {

    @Column(name = "matrikelNummer", unique = true)
    private String registrationNumber;

    private String name;

    @Embedded
    private EmbeddedScholarship scholarship;

    @Transient
    private DateTime loginTime;

…
}
```

```
@Embeddable
public class EmbeddedScholarship {

    private String description;

    private Integer amount;

}
```

Might be an issue with legacy databases, where the DB schema already exists and must not be altered.

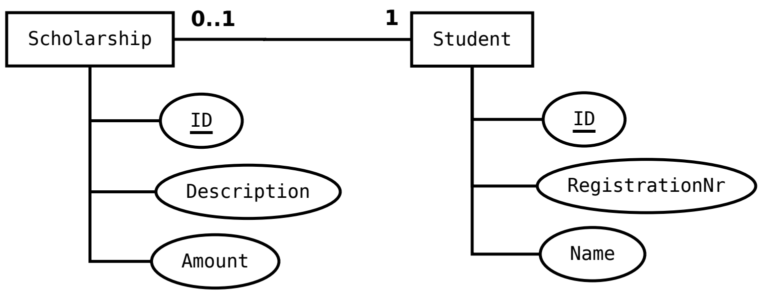
Relationship mapping

Unidirectional OneToOne using an embedded table - resulting SQL DDL

```
@Entity
public class EmbeddedStudent extends BaseEntity {

    @Column(name = "matrikelNummer", unique = true)
    private String registrationNumber;

    private String name;

    @Embedded
    private EmbeddedScholarship scholarship;

    @Transient
    private DateTime loginTime;

…
}
```

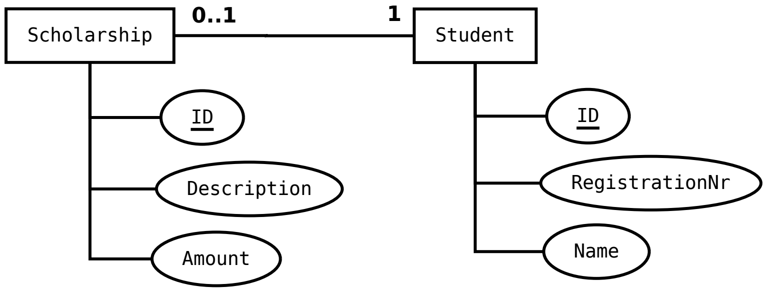
```
@Embeddable
public class EmbeddedScholarship {

    private String description;

    private Integer amount;

}
```

```
CREATE TABLE EMBEDDEDSTUDENT
(
    ID BIGINT PRIMARY KEY NOT NULL,
    NAME VARCHAR(255),
    MATRIKELNUMMER VARCHAR(255),
    AMOUNT INTEGER,
    DESCRIPTION VARCHAR(255)
);
CREATE UNIQUE INDEX anIndexName
ON EMBEDDEDSTUDENT ( MATRIKELNUMMER );
```



Relationship mapping

Bidirectional OneToOne using foreign key



The "non-owning" side

```
@Entity
public class Student extends BaseEntity {

    @Column(name = "matrikelNummer",
            unique = true)
    private String registrationNumber;

    private String name;

    @OneToOne(fetch = FetchType.LAZY,
              cascade = CascadeType.ALL,
              mappedBy="grantedTo")
    private Scholarship scholarship;

    @Transient
    private DateTime loginTime;
…
}
```

Important annotations	
@OneToOne	Defines a single-valued association to another entity that has one-to-one multiplicity.
@FetchType	LAZY = do not load referenced entity, until it is accessed for the first time EAGER = load referenced entity immediately
@CascadeType	Defines the set of cascadable operations that are propagated to the associated entity. ALL is equivalent to cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}
mappedBy	References the field that "owns" the relationship in the referenced entity. Required unless the relationship is unidirectional.

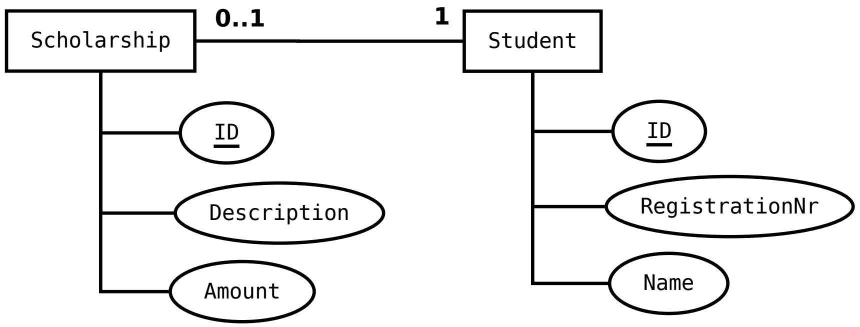
Relationship mapping

Bidirectional OneToOne using foreign key cont'd

The "owning" side

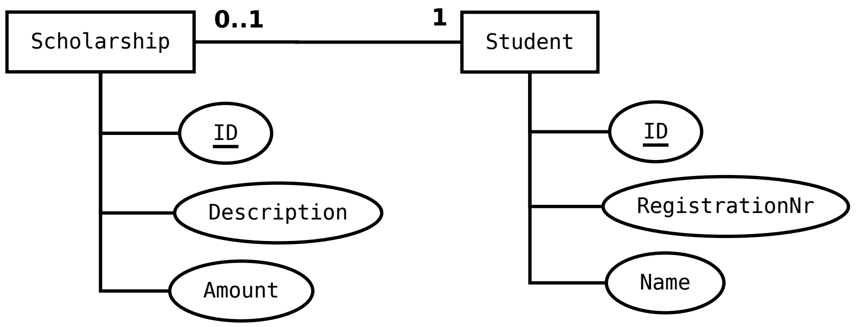
```
@Entity
public class Scholarship extends BaseEntity {

    private String description;

    private Integer amount;

    @JoinColumn(name="student_id", unique=true)
    @OneToOne
    private Student grantedTo;

…
}
```

Important annotations	
@JoinColumn	Specifies a column for joining an entity association or element collection. In this case: the name of the column, where the foreign key will be stored.



Relationship mapping

Bidirectional OneToOne using foreign key - resulting SQL DDL

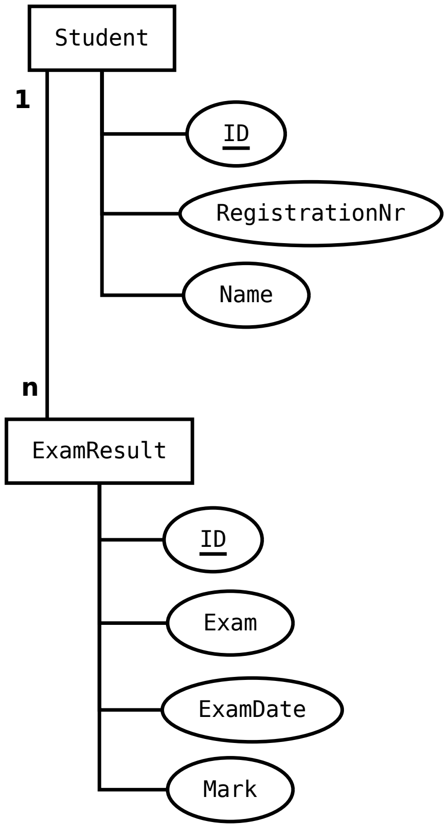


```
CREATE TABLE SCHOLARSHIP
(
    ID BIGINT PRIMARY KEY NOT NULL,
    AMOUNT INTEGER,
    DESCRIPTION VARCHAR(255),
    STUDENT_ID BIGINT,
    FOREIGN KEY ( STUDENT_ID )
    REFERENCES STUDENT ( ID )
);
CREATE UNIQUE INDEX uniqueIndexName
ON SCHOLARSHIP ( STUDENT_ID );
```

```
CREATE TABLE STUDENT
(
    ID BIGINT PRIMARY KEY NOT NULL,
    NAME VARCHAR(255),
    MATRIKELNUMMER VARCHAR(255)
);
CREATE UNIQUE INDEX anIndexName
ON STUDENT ( MATRIKELNUMMER );
```

Relationship mapping

Bidirectional OneToMany



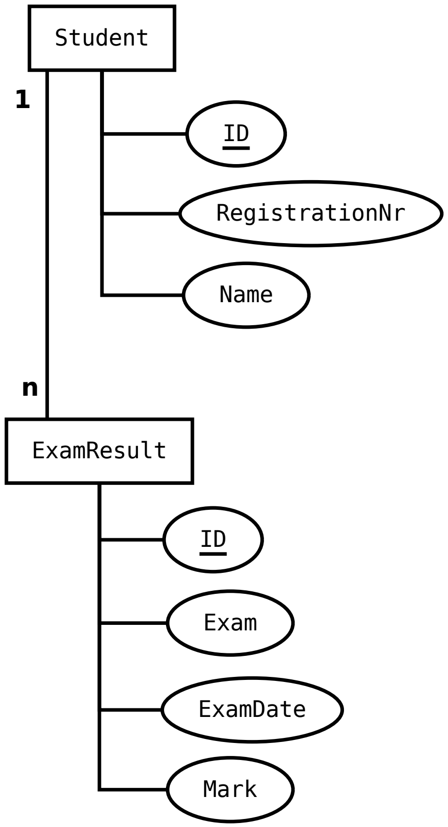
The “non-owning” side

```
@Entity
public class Student extends BaseEntity {

    @Column(name = "matrikelNummer", unique = true)
    private String registrationNumber;

    private String name;

    @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL,
              mappedBy = "grantedTo")
    private Scholarship scholarship;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "student")
    private List<ExamResult> examResults;

    @Transient
    private DateTime loginTime;
…
}
```

Important annotations	
@OneToMany	Defines a many-valued association with one-to-many multiplicity.

Relationship mapping

Bidirectional OneToMany cont'd



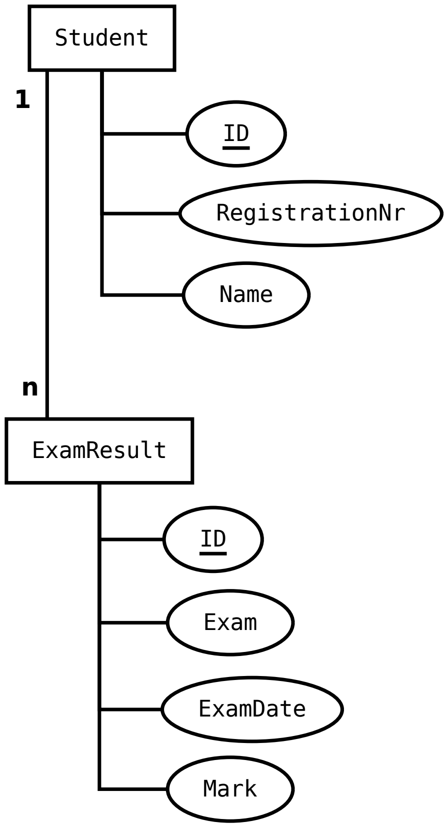
The "owning" side

```
@Entity
public class ExamResult extends BaseEntity {

    @Column(name = "prufungsDatum")
    @Temporal(TemporalType.DATE)
    private Date examDate;

    private String exam;

    private int mark;

    @ManyToOne
    private Student student;


    @Transient
    private String examLocation;
…
}
```

Important annotations	
@ManyToOne	Defines a single-valued association to another entity class that has many-to-one multiplicity.

Relationship mapping

Bidirectional OneToMany - resulting SQL DDL

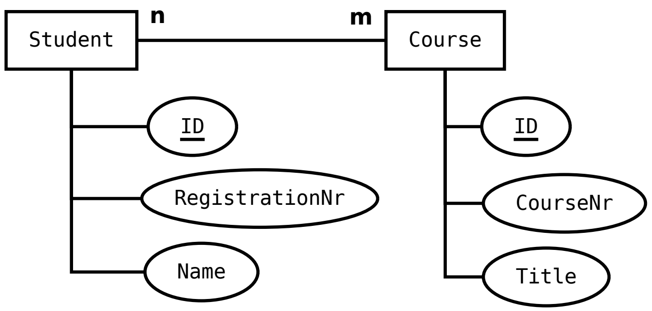


```
CREATE TABLE STUDENT
(
    ID BIGINT PRIMARY KEY NOT NULL,
    NAME VARCHAR(255),
    MATRIKELNUMMER VARCHAR(255)
);
CREATE UNIQUE INDEX anIndexNameB
ON STUDENT ( MATRIKELNUMMER );
```

```
CREATE TABLE EXAMRESULT
(
    ID BIGINT PRIMARY KEY NOT NULL,
    EXAM VARCHAR(255),
    PRUFUNGSDATUM DATE,
    MARK INTEGER NOT NULL,
    STUDENT_ID BIGINT,
    FOREIGN KEY ( STUDENT_ID ) REFERENCES STUDENT ( ID )
);
```

Relationship mapping

ManyToMany

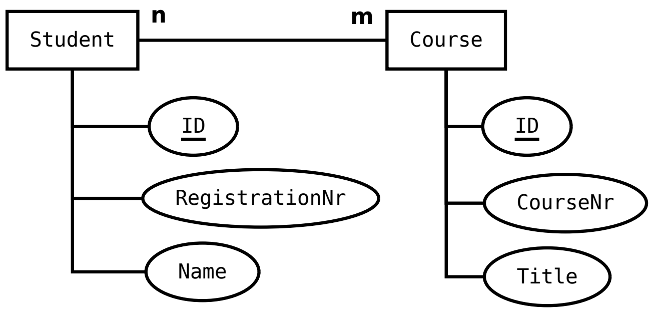


Important annotations	
@ManyToMany	Defines a many-valued association with many-to-many multiplicity.

```
@Entity
public class Student extends BaseEntity {

    @Column(name = "matrikelNummer", unique = true)
    private String registrationNumber;

    private String name;

    @OneToOne(fetch = FetchType.LAZY,
              cascade = CascadeType.ALL,
              mappedBy = "grantedTo")
    private Scholarship scholarship;

    @OneToMany(cascade = CascadeType.ALL,
               mappedBy = "student")
    private List<ExamResult> examResults;

    @ManyToMany(mappedBy = "students")
    private List<Course> courses;

    @Transient
    private DateTime loginTime;

…
}
```

Relationship mapping

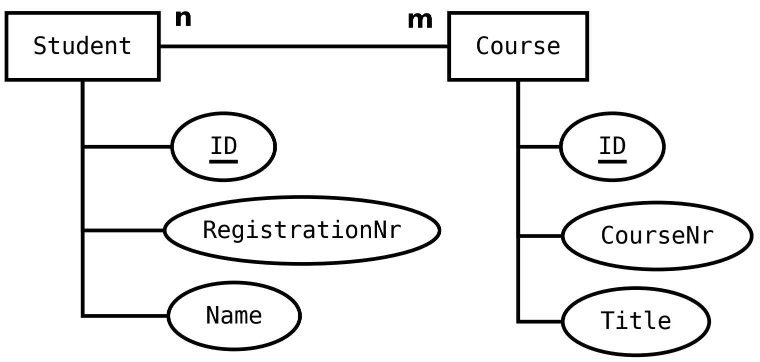
ManyToMany cont'd



```
@Entity
public class Course extends BaseEntity {

    private String courseNumber;

    private String title;

    @ManyToMany
    private List<Student> students;
…
}
```

Relationship mapping

ManyToMany - resulting SQL DDL



```
CREATE TABLE COURSE
(
    ID BIGINT PRIMARY KEY NOT NULL,
    COURSENUMBER VARCHAR(255),
    TITLE VARCHAR(255)
);
```

```
CREATE TABLE STUDENT
(
    ID BIGINT PRIMARY KEY NOT NULL,
    NAME VARCHAR(255),
    MATRIKELNUMMER VARCHAR(255)
);
CREATE UNIQUE INDEX anIndexNameB
ON STUDENT ( MATRIKELNUMMER );
```

```
CREATE TABLE COURSE_STUDENT
(
    COURSES_ID BIGINT NOT NULL,
    STUDENTS_ID BIGINT NOT NULL,
    FOREIGN KEY ( COURSES_ID )
    REFERENCES COURSE ( ID ),
    FOREIGN KEY ( STUDENTS_ID )
    REFERENCES STUDENT ( ID )
);
```

Cascade and Fetch

- Cascade Types
  - All four relationship annotations may specify operations cascaded to associated entities
  - ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH
  - Default is none
- Orphan Removal
  - For @OneToOne and @OneToMany relationships
  - Default is false
- Fetching Strategies
  - Define how object hierarchies are loaded
  - EAGER, LAZY
  - Default is eager
  - Eager must be implemented by persistence provider, lazy is optional

Persistence Concepts

- Persistence Unit (PU)
  - Defines a set of entity classes managed by the EntityManager instance in an application
  - Maps the set of entity classes to a relational database

- Persistence Context (PC)
  - Set of managed entity instances that exist in a particular data store
  - Runtime context

- Entity Manager (EM)
  - API for interaction with the persistence context
  - Manipulates and controls the lifecycle of a persistence context
  - Creates and removes persistent entity instances
  - Finds entities by its primary key
  - Runs queries on entities

Persistence Unit

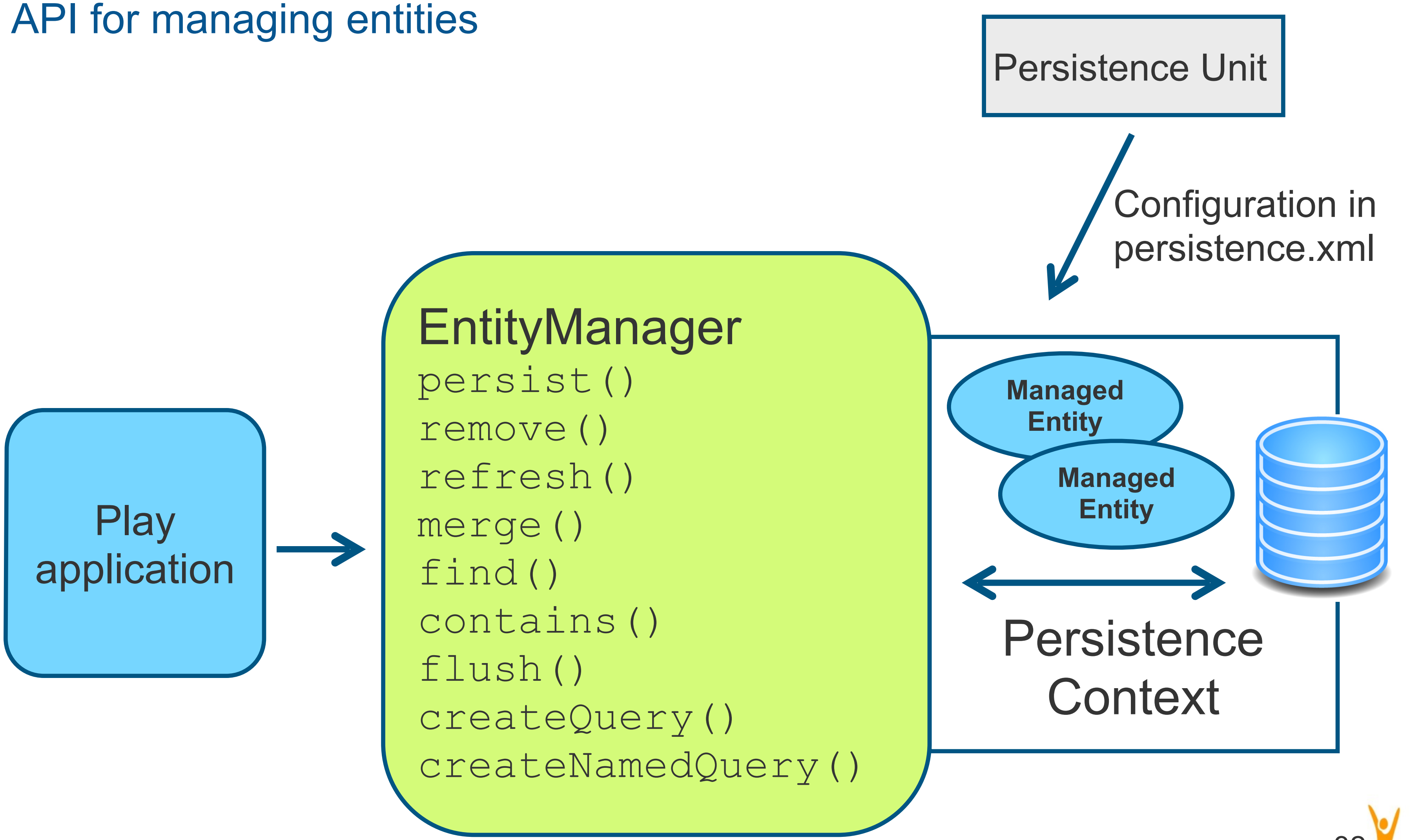
- Persistence Unit
  - Configuration to map Entity classes in an application to a relational database
- persistence.xml defines one or more persistence units
  - Defined under /conf/META-INF/persistence.xml
  - Classes with JPA annotations are automatically detected upon start of the application

Names must be referenced in application.conf

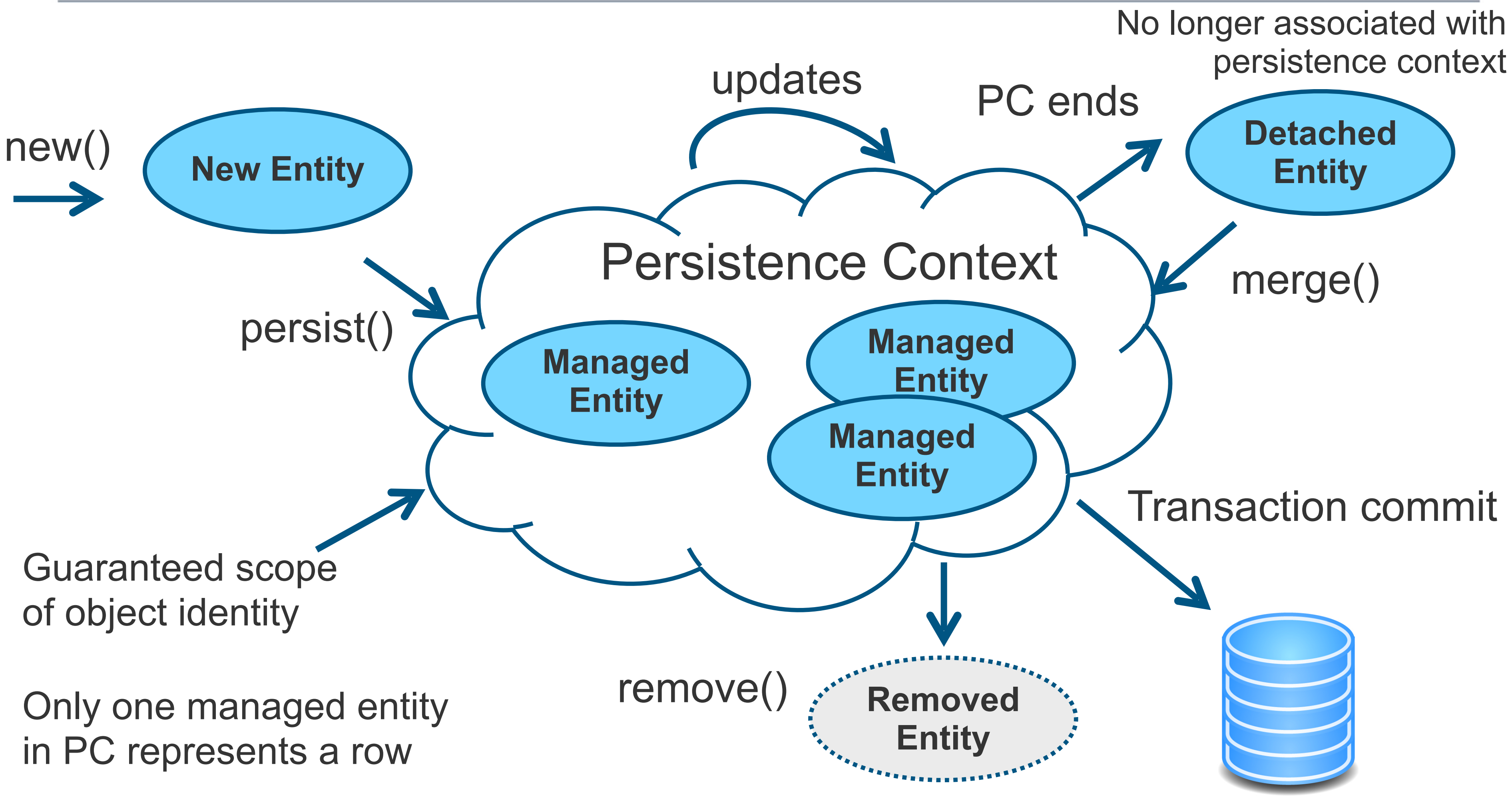
```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="defaultPersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <non-jta-data-source>DefaultDS</non-jta-data-source>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>
    </persistence-unit>
</persistence>
```

EntityManager

API for managing entities



Entity Lifecycle



Entities in managed/persistent state may be manipulated by the application and any changes will be **automatically detected and persisted** when the persistence context is flushed. There is no need to call a particular method to make your modifications persistent.

Entity states

An instance of a persistent class may be in one of three different states, defined with respect to a *persistence context* (= the Hibernate Session)

- **transient**
  The instance is not, and has never been associated with any persistence context. It has no persistent identity (= PK)
- **persistent**
  The instance is currently associated with a persistence context and has a persistent identity (= PK has a value). For a particular persistence context Hibernate guarantees, that the persistent identity is equivalent to the Java identity (= in-memory location of the object)
- **detached**
  The instance was once associated with a persistence context, but that context has been closed. It has a persistent identity and may have a corresponding row in the DB. However, Hibernate makes no guarantee about the relationship between the persistent identity and the Java identity.

Finding Entities

- Find entity by primary key using EntityManager

```
<T> T find(Class<T> entityClass, Object primaryKey)
```

- Example:

```
Student student = entityManager.find(Student.class, id);
```

- For Complex queries use
  - Java Persistence Query Language (JPQL)
  - Criteria API
  - Native Queries
- EntityManager is factory for Query objects
  - createQuery
  - createTypedQuery

JPQL

- Similar to SQL
- Portable
- Returns entities
- Select, update, delete

- Support for
    - Joins
    - Conditional Expressions
    - Functional Expressions
    - Subqueries
    - Order by, group by, having
    - …

Querying Entities with JPQL

- Dynamic Query

```
public List<Customer> findWithName (String name) {
       TypedQuery query = em.createTypedQuery (
         “SELECT c FROM Customer c”
         + “ WHERE c.name LIKE :custName”,
      Customer.class);

    query.setParameter(“custName”, name);
    query.setMaxResults(10);

    return (query.getResultList());
}
```

Querying Entities with JPQL

- Static Query
    - Named Query
    - Recommended

```
@NamedQuery(name="findAllOrders",
                query="SELECT o FROM Order o")
@Entity
Public class Order {
      ...
}

List<Order> orders =
em.createNamedQuery("findAllOrders“, Order.class)
                .getResultList();
```

Criteria API

- Alternative to JPQL, same scope
- Dynamic Queries only

- Clauses are set using Java programming language objects
  - the query can be created in a typesafe manner

- Obtain a `CriteriaBuilder` instance by using the `EntityManager.getCriteriaBuilder` method

For more information see http://docs.oracle.com/javaee/6/tutorial/doc/gjivm.html

Querying Entities with Criteria API

```
CriteriaQuery<Customer> cq =
  cb.createQuery(Customer.class);
Root <Customer> cust = cq.from(Customer.class);

// set the where clause
cq.where(cb.like(cust.get(Customer_.name), name));
cq.select(cust);
TypedQuery<Customer> q =
  entityManager.createTypedQuery(cq);
List<Customer> customers = q.getResultList();
```

References

1. Sun Microsystems. JSR 220: Enterprise JavaBeans^{TM}, Version 3.0 – Java Persitence API, 2006
2. Carol McDonald. Java Persistence API: Best Practices, Sun Tech Days 2008-2009 http://de.slideshare.net/caroljmcdonald/td09jpabestpractices2
3. Carol McDonald. Enterprise JavaBean 3.0 & Java Persistence APIs: Simplifying Persistence, Sun Tech Days 2006-2007 http://de.slideshare.net/caroljmcdonald/persistencecmcdonaldmainejug3
4. The Java EE 6 Tutorial, http://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html