

Java Persistence API

Credits to:

Piero Fraternali & Ilio Catallo

Politecnico di Milano

OBJECT MODEL VS. RELATIONAL MODEL

Object Model vs. Relational Model (1/3)

- The end-point of every web application is the DBMS
 - In many cases the DBMS has been around for much longer than the web application
- It's up to the object model of the web application to find ways to work with the database schema
 - Moving data back and forth between a DBMS and the object model is a **lot harder** than it needs to be
- Java developers find themselves spending effort in writing **lots of code** to convert row and column data into objects

Object Model vs. Relational Model (2/3)

- The technique of bridging the gap between the object model and the relational model is known as **object-relational mapping** (ORM)
- ORM techniques try to map the concepts from one model onto another
 - **Impedance mismatch**: The challenge of mapping one model to the other lies in the concepts in each for which there is no logical equivalent in the other
- We need a **mediator** to manage the automatic transformation of one to the other

Object Model vs. Relational Model (3/3)

Object Oriented Model (Java)	Relational Model
Objects, classes	Tables, rows
Attributes, properties	Columns
Identity	Primary key
Reference to other entity	Foreign key
Inheritance/Polymorphism	Not supported
Methods	Stored procedures, triggers
Code is portable	Not necessarily portable (depending on the vendor)

The Problem with JDBC

- **JDBC** has been the first major support for database persistence
- JDBC offers an **abstraction** of the proprietary client programming interfaces offered by database vendors
 - It allows Java programs to fully interact with the database
- **Problem:** JDBC is portable, but the SQL language **is not**
 - The burden of conversion between relational to OO is on the programmer

A bit of reading

- Ted Neward: «The Vietnam of Computer Science»
- <http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx>

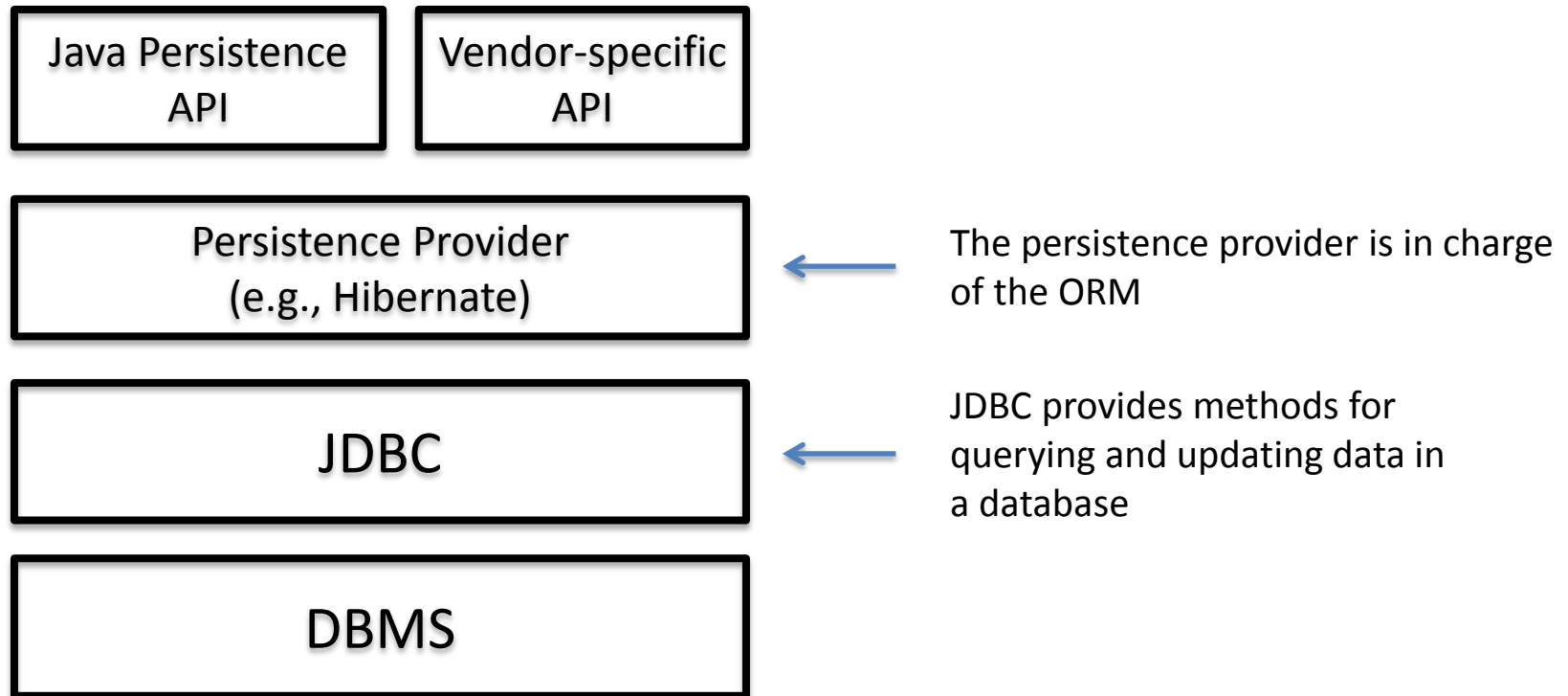
Java Persistence API

- The **Java Persistence API** bridges the gap between object-oriented domain models and relational database systems
 - JPA provides a POJO (Plain Old Java Object) persistence model for object-relational mapping
- Adds up to other previous proposals
 - JDO: Java data Objects (JDO 2.0: JSR 243)
 - JDBC: Java database connectivity (JDBC 3.0 API)
- Developed as part of [JSR-317](#)
 - In addition to support within EJB, JPA can be used in a standalone Java SE environment
 - Usable with / without a container

Java Persistence API

- The Java Persistence API can automatically map Java object **to and from** a relational database
 - Objects can be synchronized with an underlying persistent storage provider
- Persistence provides an ease-of-use abstraction **on top** of JDBC
 - The code may be isolated from the DB and vendor-specific peculiarities

JPA Architecture



JPA in a nutshell

- Java Persistence API 2.0 main features:
 - **POJO Persistence:** there is nothing special about the objects being persisted, any existing non-`final` object with a default constructor can be persisted
 - **Non-intrusiveness:** the persistence API exists as a separate layer from the persistent objects, i.e., the persisted objects are not aware of the persistence layer
 - **Object queries:** a powerful query framework offers the ability to query across entities and their relationships without having to use concrete foreign keys or database columns

JPA main concepts

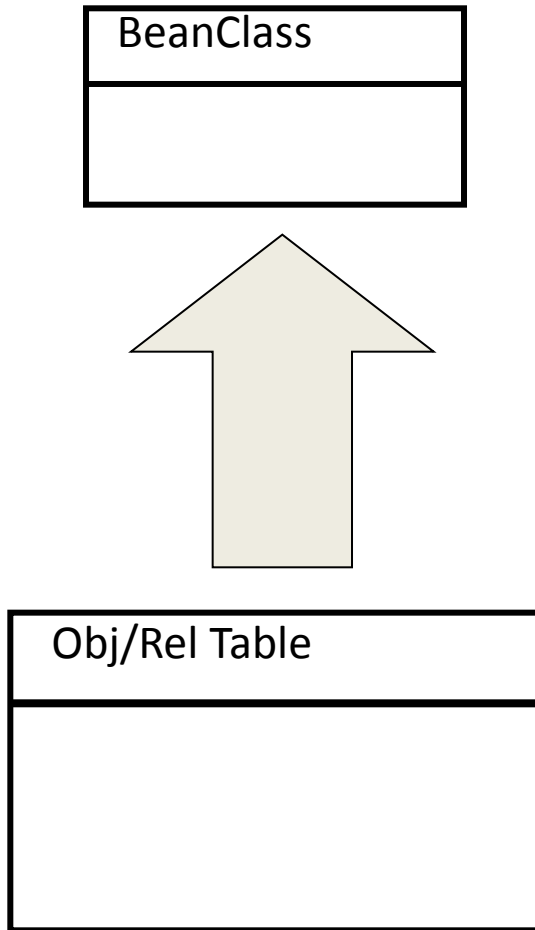
- **Entity**: a class (JavaBean) representing a set of persistent objects mapped onto a relational table
- **Persistence Unit**: the set of all **classes** that are persistently mapped to **one** database (analogous to the notion of **db schema**)
- **Persistence Context**: the set of all **objects** of the entities defined in the persistence unit (analogous to the notion of **db instance**)
- **Entity manager**: the interface for interacting with a Persistence Context

ENTITY AND ENTITYMANAGER

Entity

- A Java Bean (POJO Plain Old Java Object) that represents a table in a database
- The class represents the **table**
- The objects represent the **tuples**
- May have a life longer than that of the application
- Needs to be associated with the database table it represents (**mapping**)
- Inherits much of the properties & concepts of an object-relational database table

Entity Properties



- Identification (primary key)
- Nesting
- Relationship
- Inheritance
- Referential integrity (foreign key)

Entity (example)

Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    public Employee() {}
    public Employee(int id) { this.id = id; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary(long salary) { this.salary = salary; }
}
```

Annotations are used to qualify the class as an entity

The entity instance is just a POJO

Entity constraints

- Entities must respect the following requirements:
 - The entity class must have a `public` or `protected` no-arg constructor
 - The entity class must not be `final`
 - No method or persistent interface variables of the entity class may be `final`
 - If an entity instance is to be passed by value as a detached object, the `Serializable` interface must be implemented
- The persistent state of an entity is represented by instance variables, which may correspond to JavaBean properties

Entity Identification

- In database, objects and tuples have an identity (**primary key**)
- → an entity bean assumes the identity of the persistent data it is associated to
- **Simple** Primary key = persistent field of the bean used to represent its identity
- **Composite** Primary key = set of persistent fields used to represent its identity
- **Remark**: with respect to the session bean, the PK is a new concept. Session beans do not have a durable identity

Entity identification syntax

@Entity

```
public class Book {
```

@Id

```
private String isbn;  
private String title;  
private int pages;  
}
```

- @Id tags a field as the simple primary key
- Composite primary keys are denoted using the **@EmbeddedId** and **@IdClass** annotations

Entity: Identifier generation

- Sometimes, applications do not want to explicitly manage uniqueness in some aspect of their domain model
 - The persistence provider can automatically generate an identifier for every entity instance of a given type
- This persistence provider's feature is called **identifier generation** and is specified by the `@GeneratedValue` annotation

Identifier generation options

- Applications can choose one of four different ID generation strategy

Id generation strategy	Description
AUTO	The provider generates identifiers by using whatever strategy it wants
TABLE	Identifiers are generated according to a generator table
SEQUENCE	If the underlying DB supports sequences, the provider will use this feature for generating IDs
IDENTITY	If the underlying DB supports primary key identity columns, the provider will use this feature for generating IDs

Identifier generation annotation

Using AUTO identifier generation strategy

```
@Entity
public class Employee {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    ...
}
```

- Remember that the AUTO mode is a generation strategy mainly meant for developing prototypes
- In any other situation, it would be better to use other generation strategies

Mapping annotations

```
@Entity @Table(name="T_BOOKS")
public class Book {
    @Column(name="BOOK_TITLE", nullable=false)
    private String title;
    @Enumerated(EnumType.STRING)
    private CoverType coverType;
    @Temporal(TemporalType.DATE)
    private Date publicationDate;
    @Transient
    private BigDecimal discount;
}
```

Field Properties

- All fields of an entity are persistent, unless otherwise specified with the **@transient** annotation
- Persistent fields **can** be mapped by annotations onto the relational schema of the underlying database
- **Alternatively all mapping information can be provided in one configuration file (XML descriptor)**
- The XML descriptor is both an **alternative** to and an **overriding** mechanism for annotations (the descriptor overrides the annotations in the bean code)

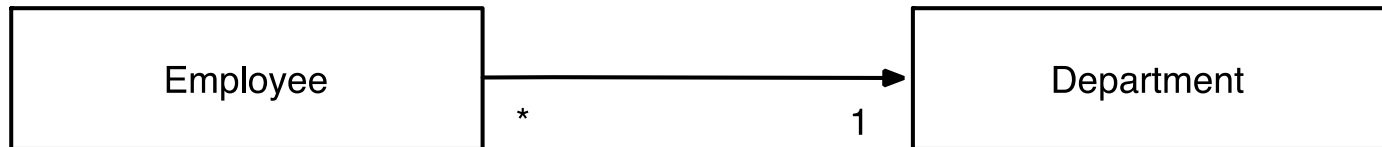
Property-style annotation

- If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components
 - `Type getProperty()`
 - `void setProperty(Type type)`
 - `Set<Type> getMultiValueProperty () {}`
 - `void setMultiValueProperty(Set<Type>) {}`
- Annotations can be placed on the getter method

MAPPING RELATIONSHIPS

Entities and relationships

- If entities contained only simple persistent state, the issue of ORM would be a trivial one
- In fact, most entities need to be able to have **relationships** with other entities
 - This is what produces the **domain model** graph for the application



Relationship's features: Overview

- Every relationship has four characteristics:
 - **Cardinality**: the number of entity instances that exist on each side of the relationship
 - **Directionality**: each of the two entities may have an attribute that points to the related entity
 - **Role**: each entity in the relationship is said to play a role
 - **Ownership**: one of the two entity in the relationship is said to own the relationship

Relationship's features:

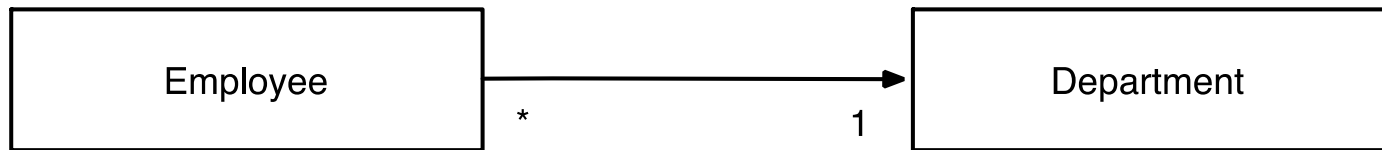
Directionality

- Each entity in the relationship may have a **pointer** to the other entity:
 - When each entity points to the other, the relationship is **bidirectional**
 - If only one entity has a pointer to the other, the relationship is said to be **unidirectional**
- All relationships in JPA are unidirectional
 - A bidirectional relationship have to be intended as a pair of unidirectional mappings

Relationship's features:

Roles

- Depending on directionality, we can identify the entity playing the role of **source** and the entity playing the role of **target**



- Employee and Department are involved in a unidirectional relationship
 - Employee is the source entity
 - Department is the target entity

Relationship's features:

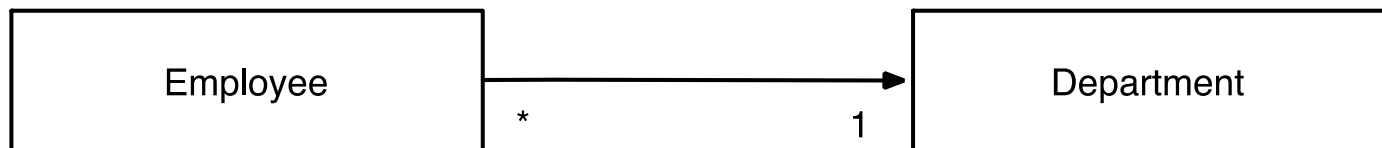
Cardinality

- Each role in the relationship will have its own cardinality. This leads to four possible combinations:
 - **Many-to-one:** many source entities, one target entity
 - **One-to-many:** one source entity, many target entities
 - **One-to-one:** one source entity, one target entity
 - **Many-to-many:** many source entities, many target entities
- Remember: bidirectional relationships are just pairs of unidirectional relationships with swapped source and target entities

Relationship's features:

Ownership (1/2)

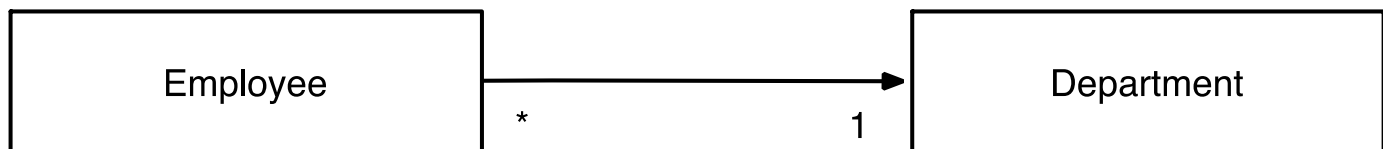
- In the database, relationships are implemented by introducing a column that refers to a key in another table
 - In JPA terminology, such a column is called **join column**
- Example:
 - Many-to-one unidirectional relationship between Employee and Department
 - The underlying Employee table has a join column containing the Department primary key



Relationship's features:

Ownership (2/2)

- In almost every relationship, one of the two entities will have the join column in its table
 - This entity is called the **owner** of the relationship and its side is called the **owning side**
- Example: Employee is the owner of the relationship



Many-to-one mappings

- In a many-to-one mapping the owner of the relationship is the source entity
- A many-to-one mapping is defined by annotating the source entity with the `@ManyToOne` annotation

`@ManyToOne` annotation in `Employee.java`

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToOne
    private Department department;
    ...
}
```

One-to-many mappings

- In a one-to-many mapping the owner of the relationship is the target entity
- Therefore, the `@OneToMany` annotation must come with the `mappedBy` element
 - The `mappedBy` element indicates that the owning side resides on the other side of the relationship

@OneToMany annotation in Department.java

```
@Entity
public class Department {
    @Id private int id;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    ...
}
```

← The attribute on the target entity that owns the relationship

One-to-one mappings (1/2)

- In a one-to-one mapping the owner can be either the source entity or the target entity
- A one-to-one mapping is defined by annotating the owner entity with the `@OneToOne` annotation

`@OneToOne` annotation in `Employee.java`

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne
    private ParkingSpace parkingSpace;
    ...
}
```

One-to-one mappings (2/2)

- If the one-to-one mapping is bidirectional, the inverse side of the relationship needs to be specified too
- In the non-owner entity, the `@OneToOne` annotations must come with the `mappedBy` element

`@OneToOne` annotation (inverse side) in `ParkingSpace.java`

```
@Entity
public class ParkingSpace {
    @Id private int id;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    ...
}
```

Many-to-many mappings (1/2)

- In a many-to-many mapping there is no join column
 - The only way to implement such a mapping is by means of a **join table**
- Therefore, we can arbitrarily specify as owner either the source entity or the target entity

@ManyToMany annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToMany
    private Collection<Project> projects;
    ...
}
```

Many-to-many mappings (2/2)

- If the many-to-many mapping is bidirectional, the inverse side of the relationship needs to be specified too
- In the non-owner entity, the `@ManyToMany` annotation must come with the `mappedBy` element

`@ManyToMany` annotation (inverse side) in `Project.java`

```
@Entity
public class Project {
    @Id private int id;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    ...
}
```

Lazy Loading (1/4)

- When loading an entity it is questionable if related entities are to be fetched & loaded too
 - Performance can be optimized by defer fetching data until the time when they are needed
- This design pattern is called **lazy** (opposite=**eager**) loading
 - At relationship level, lazy loading can be of great help in enhancing performance because it can reduce the amount of SQL that is executed
- Loading policy can be expressed specifying **fetch mode** for relationships

Lazy Loading (2/4)

- When the **fetch mode** is not specified:
 - On a single-valued relationship, the related object **is guaranteed** to be loaded eagerly
 - Collection-valued relationships **default** to be lazily loaded
- In case of bidirectional relationships, the fetch mode might be lazy on one side but eager on the other
 - Quite common situation, relationships are often accessed in different ways depending on the direction from which navigation occurs

Lazy Loading (3/4)

- The directive to lazily fetch an attribute is meant only to be a **hint** to the persistence provider
 - The provider is not required to respect the request because the behavior of the entity will not be compromised if the provider decides to eagerly load data
- The converse is **not true** because specifying that an attribute be eagerly fetched might be critical to access the entity once detached

Lazy Loading (4/4)

Lazy loading of the `parkingSpace` attribute

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    ...
}
```

- The fetch mode can be specified on any of the four relationship mapping types
- The `parkingSpace` attribute will not be loaded immediately after the `Employee` is loaded, but only when actually accessed

Cascading operations (1/4)

- By default, every `EntityManager`'s operation applies only to the entity supplied as an argument to the operation
 - The operation **will not** cascade to other entities that have a relationship with the entity that is being operated on
- For some operations (e.g., `remove()`) this is usually the **desired behavior**

Cascading operations (2/4)

- Some other operations usually require cascading, such as `persist()`
 - If an entity has a relationship to another entity normally the two must be persisted **together**
- Example: Many-to-one unidirectional mapping between `Employee` and `Address`

Manually cascading

```
Employee emp = new Employee();  
Address addr = new Address();  
emp.setAddress(addr);  
em.persist(addr);  
em.persist(emp);
```

← We would like to avoid explicit persisting the `Address` entity instance

Cascading operations (3/4)

- JPA provides a mechanism to define when operations such as `persist()` should be automatically cascaded across relationships

Enabling cascade persist

```
@Entity
public class Employee {
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
}
```

- ▣ You need to be sure that the `Address` instance has been set on the `Employee` instance before invoking `persist()` on it

Cascading operations (4/4)

- The `cascade` attribute accepts several possible values coming from the `CascadeType` enumeration:
 - `PERSIST`, `REFRESH`, `REMOVE`, `MERGE` and `DETACH`
 - The constant `ALL` is a shorthand for declaring that all five operations should be cascaded
- As for relationships, cascade settings are unidirectional
 - they must be explicitly set on both sides of a relationship if the same behavior is intended for both situations

Mapping Inheritance

- Inheritance can be used also for persistent objects, for factoring out data members inherited by multiple subclasses
- Hierarchies of entities can be defined
- The mapping of a hierarchy to the database can follow different strategies:
 - Single table per hierarchy
 - Table per class
 - Joined
- Hierarchies can mix persistent entities and transient Java classes and involve abstract (i.e., non instantiable) entity classes

Hierarchy: syntax

@MappedSuperclass abstract class

```
Publication {  
    @Id private Long id;  
    @ManyToOne private Editor editor;  
}
```

@Entity public class Book **extends Publication**

```
{  
    @Temporal(TemporalType.DATE)  
    private Date pubDate;  
}
```

@Entity public class Comic **extends**

```
Publication {  
    @Enumerated(EnumType.STRING)  
    private IssuePeriod issuePeriod;  
}
```

- Mapped superclasses are a syntactic facility to share persistent fields and mapping information, they do not have entities
- Alternatively, the superclass can also be an entity or a transient Java class

Single Table per Hierarchy

- A discriminator column must be used to distinguish the type of the object

`@Inheritance(strategy=SINGLE_TABLE)`

`@DiscriminatorColumn(name="Discr")`

`@MappedSuperclass` abstract class Publication { . }

`@DiscriminatorValue("B")`

`@Entity` class Book extends Publication { . }

`@DiscriminatorValue("C")`

`@Entity` class Comic extends Publication { . }

Publication				
id	Discr	editor_id	publDate	issuePeriod
1	C	1	null	MONTHLY
2	B	1	2007/10/20	null

Joined

- One table per class (including superclasses)
- Objects reconstructed by join

`@Inheritance(strategy=JOINED)`

`@MappedSuperclass` abstract class Publication { . }

`@Entity` class Book extends Publication { . }

`@Entity` class Comic extends Publication { . }

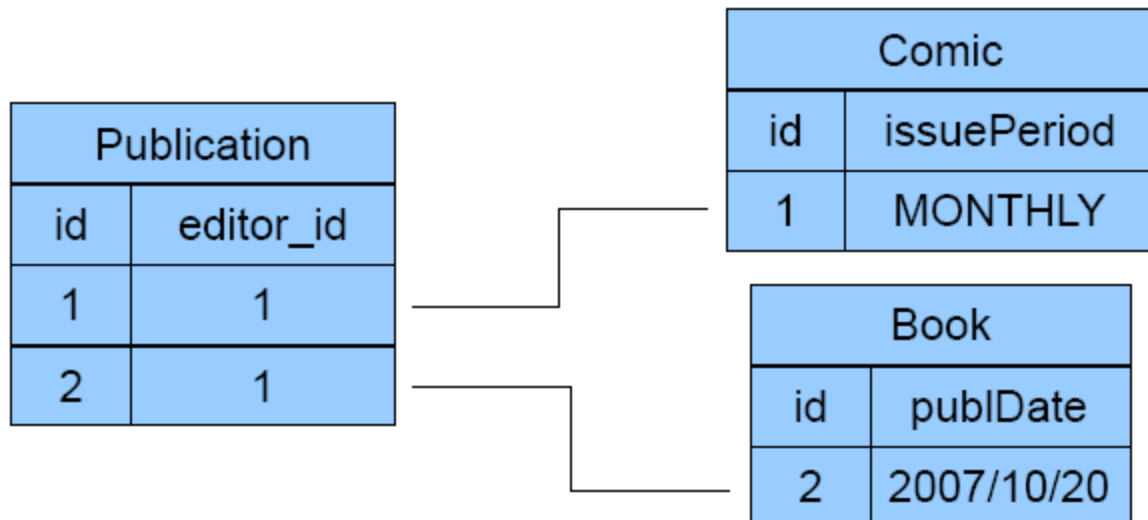


Table per Class

- One table per **concrete** class
- Inherited attributes **cached** in subclasses
- Population reconstructed by UNION

@Inheritance(strategy=TABLE_PER_CLASS)

@MappedSuperclass abstract class Publication { . }

@Entity class Book extends Publication { . }

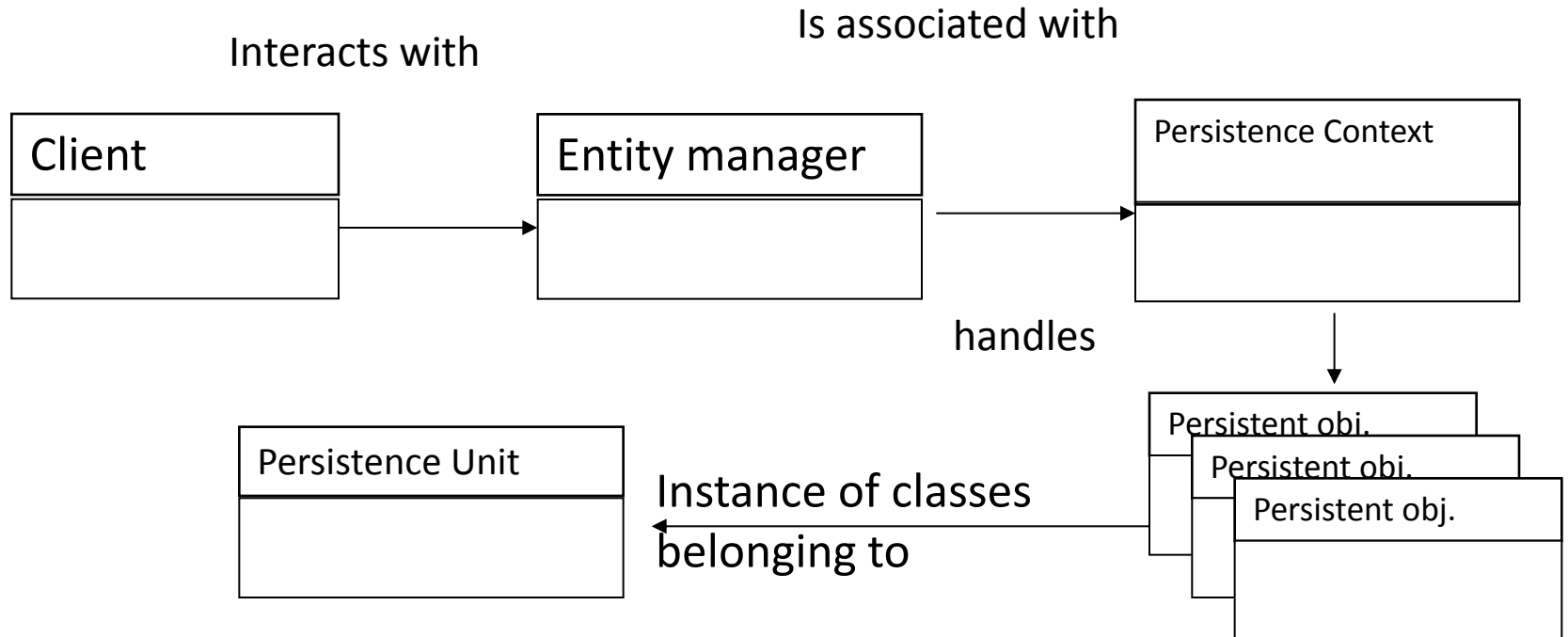
@Entity class Comic extends Publication { . }

Comic		
id	editor_id	issuePeriod
1	1	MONTHLY

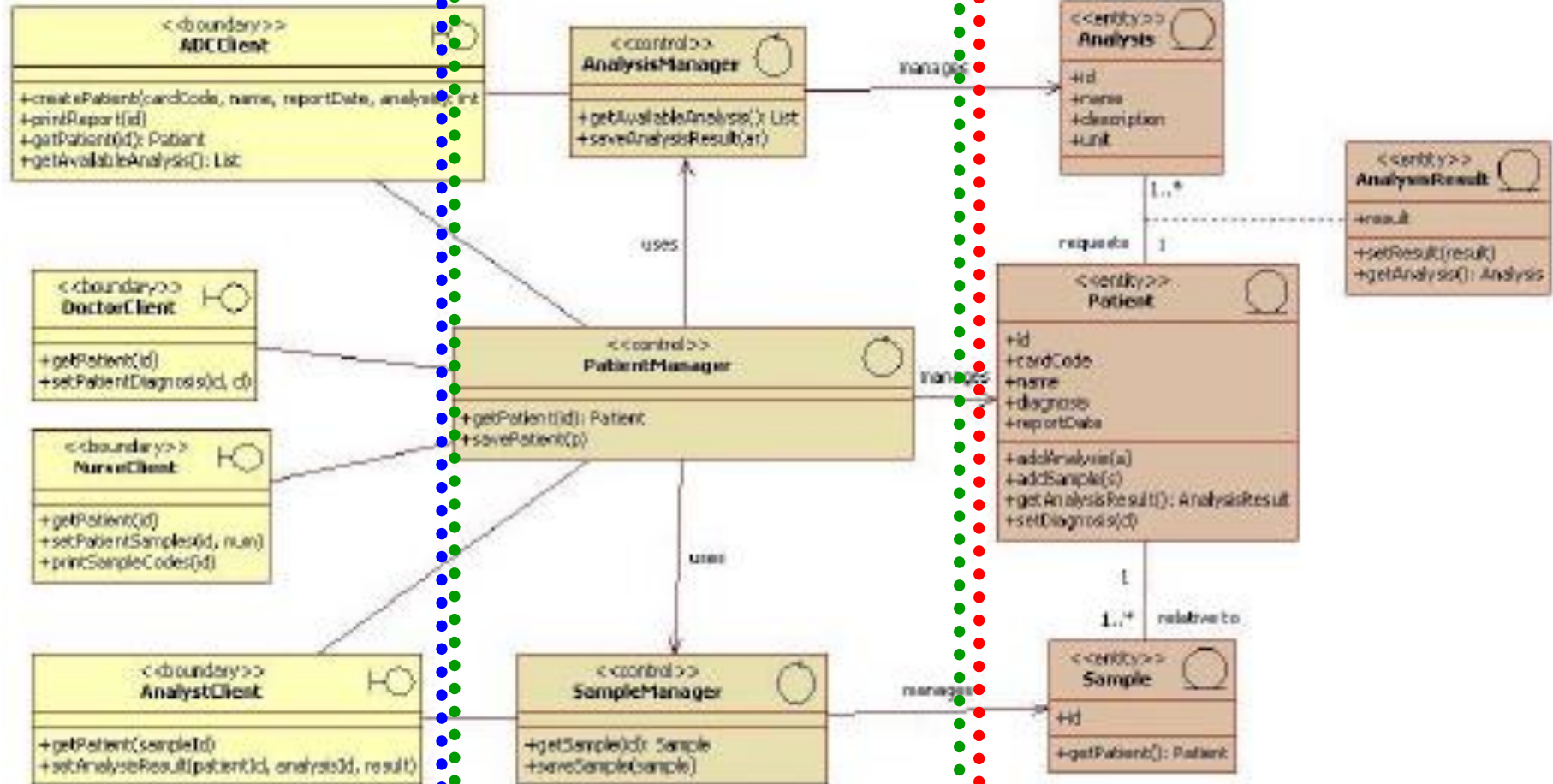
Book		
id	editor_id	publDate
2	1	2007/10/20

How to work with entities

- Entities are accessed through suitable interfaces of JPA



Typical project structure in JEE



Web tier

Session Beans

Entity Beans

EntityManager

- Because entity instances are plain Java objects, they do not become persistent until the application invoke an API method to initiate the process
- The **EntityManager** is the central authority for all persistence actions
 - It manages the ORM between a fixed set of entity classes and an underlying data source
 - It provides APIs for creating queries, finding objects, synchronizing, and inserting objects into the DB

EntityManager Interface

Method signature	Description
public void persist (Object entity);	Persists an entity instance in the database
public <T> T find (Class<T> entityClass, Object primaryKey);	Finds an entity instance by its primary key
public void remove (Object entity);	Removes an entity instance from the database
public void refresh (Object entity);	Resets the entity instance from the database
public void flush ();	Synchronizes the state of entities with the database

Creating a new POJO

Creating a new POJO

```
Employee e = new Employee(ID, "John Doe")
```

- Calling the `new` operator **does not** magically interact with some underlying service to create the `Employee` in the database
 - Instances of the `Employee` class remain POJOs until you ask the `EntityManager` to persist its state in the database
- When an entity is first instantiated, it is in the **transient** (or **new**) state since the `EntityManager` does not know it exists yet

Persisting an entity

Persisting an entity

```
Employee emp = new Employee(ID, "John Doe");  
em.persist(emp);
```

- `EntityManager`'s `persist()` method creates a new record in the database corresponding to the entity
- The entity enters the **managed** state, i.e., the `EntityManager` makes sure that the entity's data is synchronized with the database

Finding an entity

Finding an entity

```
Employee emp = em.find(Employee.class, ID);
```

- `EntityManager`'s `find()` method takes as an input the class of the entity that is being sought and the primary key that identifies the entity
- When the call completes, the returned `Employee emp` will be a managed entity
 - If the entity was not found, then the `find()` method will return `null`

Removing an entity

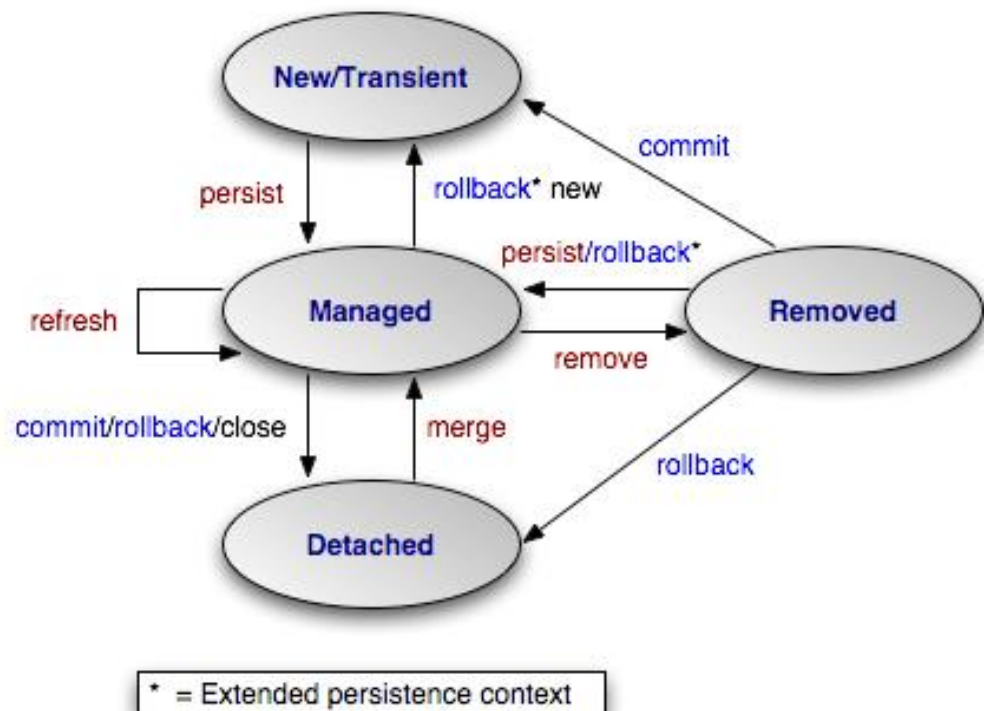
Removing an entity

```
em.remove(emp);
```

- `EntityManager`'s `remove()` method removes the data associated with the entity from the database
- The `remove()` method will **detach** the entity, i.e., the entity is no longer managed by the `EntityManager`
- There is no guarantee that the state of the entity is in synch with the database

Entity's Lifecycle

- **NEW**: No persistent identity
- **MANAGED**: Associated with persistence context, changes to objects automatically synch to db (**NOT VICE VERSA**)
- **DETACHED**: Has persistent identity but changes are NOT automatically propagated to db
- **REMOVED**: Scheduled for removal from the db
- **DELETED**: erased from db



Typical workflow with entities

- The client normally proceeds as follows:
 - {
 - \\ locate an EntityManager instance
 - \\ find or create entity instances
 - \\ manipulate entities and relationships
 - \\ persist changes
 - } transaction policy depends on the Entity Manager

Refreshing an entity

- Applied to an entity in the managed state
- Aligns the value of data members with the current state of the corresponding tuple of the database, discarding any changes made in main memory;
- Updates both the object passed as an actual parameter and those related through relationships with `cascadeType.REFRESH` or `cascadeType.ALL`.

Detaching an entity

- The instance is no longer associated with a `PersistenceContext`
- Changes are no longer automatically written in the database
- Its status at the time of detachment remains accessible to the application
 - member data with fetch policy different from LAZY;
 - member data previously accessed by the application;
 - related objects, whether they are already extracted from the database previously, due to a query or a fetch strategy of type EAGER.

How detachment occurs

- **explicitly**
 - for a specific instance, with the direct or the cascade invocation of the detach() method;
 - for all instances of the PersistenceContext, with the invocation of the clear() and close() methods
- **implicitly**
 - when the object is serialized, for example by passing it to an application module that resides on a different network node
 - upon transactional events: transaction rollback, or commit when using an EntityManager that automates the management of transactions

Merging an instance

- `merge()` propagates state changes from a detached instance to a managed instance
- The method works by **copying** member data values from the detached instance into the data members of an existing instance having the same identity, or,
 - if such an instance does not exist, a new one is created to allow the copy
- `merge()` is propagated recursively along relationships with annotation `cascadeType.MERGE`

Persistence Unit & Persistence Context

- An `EntityManager` maps a fixed set of classes (i.e., entities) to a particular database. This set of entities is called **persistence unit**
 - Each persistence unit is tied to **one and only one** data source
 - Each persistence unit is defined in the deployment descriptor `CLASSPATH/META-INF/persistence.xml`
- Given a persistence unit, the set of managed entity instances is called **persistence context**

Persistence Unit: persistence.xml

persistence.xml

```
<persistence>
  <persistence-unit name="EmployeeUnit">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>it.polimi.awt.jpa.Employee</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://serverurl:3306/db-schema"/>
      <property name="javax.persistence.jdbc.user"
        value="dbuser"/>
      <property name="javax.persistence.jdbc.password"
        value="dbpasswd"/>
    </properties>
  </persistence-unit>
</persistence>
```

the JPA persistence provider to be used

Multiple <class> elements can be specified when there is more than one entity

The names of the JDBC properties were standardized in JPA 2.0

Locating an EntityManager in EJB

- From within an EJB session bean, using dependency injection

```
@Stateless
public class SessionEJB {
    @PersistenceUnit
    private EntityManagerFactory factory;
    @PersistenceContext
    private EntityManager manager;
}
```

Obtaining an EntityManager in JSE

- An EntityManager object is created from a EntityManagerFactory
 - As with JDBC, in which a Connection is created from a DriverManager

Obtaining an EntityManager object in JSE environment

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("EmployeeUnit");  
EntityManager em = emf.createEntityManager();
```

- Persistence is a bootstrap class that is used to obtain an EntityManagerFactory in Java SE environments

Managing the EntityManager

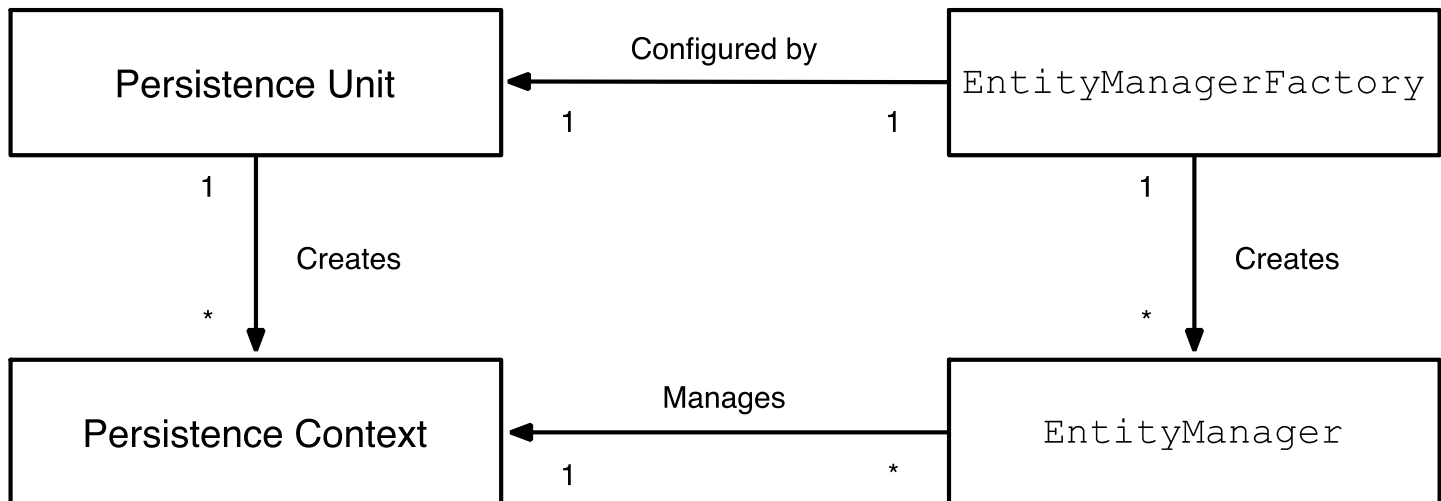
- In a JSE environment, developers are responsible for writing code to control **every aspect** of the EntityManager's lifecycle
 - EntityManager keeps managing attached entities until they are closed
 - Resources need to be explicitly released

Releasing resources

```
entityManager.close();  
entityManagerFactory.close();
```

Working with multiple EntityManager

- For each persistence unit there is an `EntityManagerFactory`
- Many `EntityManager`s can point to the same persistence context



Example: JPA calls from an EJB

```
@Stateless public class SessionEJB {  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    public void createBook(String isbn, Long editorId, ...) {  
        Editor editor = entityManager.find(Editor.class, editorId);  
        Book b = new Book(isbn, editor, ...);  
        entityManager.persist(b);  
    }  
}
```

JPQL & Query API

QUERYING THE DATA SOURCE

Java Persistence Query Language

- The **Java Persistence Query Language (JPQL)** is a platform-independent object-oriented query language defined as part of the JPA specification
- The main difference with SQL is that:
 - JPQL operates on classes and objects (entities)
 - SQL operates on tables, columns and rows
- Therefore, the result set of a JPQL query is a **collection of entities**, rather than a list of tuples

JQPL statements

- JPQL supports three types of statements:
 - `SELECT` statements retrieve entities or entity-related data
 - `UPDATE` statements update one or more entities
 - `DELETE` statements delete one or more entities
- JPQL syntax looks like SQL, but JPQL **is not** SQL

Example of `SELECT` statement

```
SELECT i FROM Item i
```

JQPL: FROM clause

- The FROM clause of JPQL defines the **domain** for the query, i.e., the names for the entities that will be used in the query

Example of SELECT statement

```
SELECT i FROM Item i
```


- ▣ `Item` is the domain that we want to query, and `i` is intended as an **identifier** of type `Item`
- ▣ Identifier variables can be used in other clauses of the same query (e.g., WHERE)

JQPL: Path Expressions

- Given an identifier variable, we can use dot notation to access a specific field of the entity
 - Such expressions are known as **path expressions**
- Path expressions are normally used to:
 - narrow the domain for a query by using it in a `WHERE` clause
 - order the retrieved result by using a `ORDER BY`

Example of path expression

```
SELECT c  
FROM Category c  
WHERE c.items is NOT EMPTY
```

 `c.items` is a collection, hence this expression is a **collection-value** path expression

JPQL: JOIN operator

- In many common situation there is the need to join two or more entities based on their relationships
- Joins can be specified either in the `WHERE` clause or in the `FROM` clause by means of the `JOIN` operator

Join operation in the `WHERE` clause

```
SELECT p.number  
FROM Employee e, Phone p  
WHERE e = p.employee  
AND  
e.department.name = 'NA42'
```

Join operation in the `FROM` clause

```
SELECT p.number  
FROM Employee e JOIN e.phones p  
WHERE e.department.name =  
    'NA42'
```

Querying the data source

- With JPA, developers can use the following methods to retrieve entities and related data:
 - `EntityManager.find()` with entity's primary key
 - Queries written in Java Persistence Query Language (JPQL)
 - Queries written in SQL native to the underlying database
- Both JPQL and SQL queries are executed by means of the **Query API**

Query API: Overview

- The required steps to create a JPA query are similar to those of a traditional JDBC query

Basic steps for JDBC Query using SQL	Basic steps for a JPA Query using JPQL
Obtain a database connection	Obtain an instance of an <code>EntityManager</code>
Create a query statement	Create a query instance
Execute the statement	Execute the query
Retrieve the results (DB tuples)	Retrieve the results (Collection of entities)

Query API:

Named & dynamic queries (1/2)

- The Query API supports two type of queries:
 - **Named queries** are intended to be stored and reused
 - **Dynamic queries** are created and executed on the fly
- The first step to execute named or dynamic query is to create a **query instance**
 - The `EntityManager` interface provides methods for creating query instance

Query API:

Named & dynamic queries (2/2)

Method signature	Description
public Query <code>createQuery</code> (String qlString);	Creates a dynamic query using a JPQL statement
public Query <code>createNamedQuery</code> (String name);	Creates a query instance based on a named query

Query API:

Named query (1/2)

- Any query that is used in multiple components of your application is a candidate for a named query
 - They can enhance performance because they are prepared once and can be efficiently reused
- Named queries are stored **on the** entity

Defining a named query

```
@Entity
@NamedQuery(
    name = "findAllCategories",
    query = "SELECT c FROM Category c WHERE c.categoryName
            LIKE :categoryName ")
public class Category {
    ..
}
```

Query API:

Named query (2/2)

- Named queries are **globally scoped**, i.e., named query instances can be created from any component that has access to the persistent unit
 - As a consequence, a named queries must have a unique name in the whole persistence unit
- Named query are instantiated by name

Instantiating a named query

```
Query query = em.createNamedQuery("findAllCategories");
```

Query API:

Dynamic query

- A dynamic query can be created wherever the `EntityManager` is available
- The only requirement is to pass a valid JPQL statement

Instantiating a dynamic query

```
Query query = em.createQuery("SELECT i FROM Item i");
```

Query API:

Executing queries

- The `Query` interface defines several methods for executing a query

Method signature	Description
public Query <code>setParameter(int position, Object value);</code>	Sets the value for a positional parameter
public Query <code>setParameter(String name, Object value);</code>	Sets the value for a named parameter
public List <code>getResultList();</code>	Retrieves a result set for a query
public Query <code>setMaxResults(int maxResult);</code>	Sets the maximum number of objects to be retrieved

Query API:

Parameters (1/2)

- The number of entities retrieved in a query can be limited by specifying a `WHERE` clause
 - `WHERE` clauses can be parameterized
- Two ways of specifying parameters:
 - By number (positional parameter)
 - By name (named parameter)
- Before executing a query, all parameters need to be set
 - This can be done by using the `Query's setParameter()` method

Query API: Parameters (2/2)

Executing query instance with positional parameters

```
Query query = em.createQuery("SELECT i FROM Item i WHERE  
                             i.initialPrice = ?1");  
query.setParameter(1, 100.00);  
List<Item> items = query.getResultList();
```

Executing query instance with named parameters

```
Query query = em.createQuery("SELECT c FROM Category c WHERE  
                             c.categoryName LIKE  
                             :categoryName");  
query.setParameter("categoryName", categoryName);  
List<Category> categories = query.getResultList();
```

Criteria API

QUERYING THE DATA SOURCE

Criteria API: Overview

- The Criteria API is especially suitable to build complex, dynamic queries where the structure of the criteria is not known until **runtime**
- The Criteria API also provides a way of creating **type-safe queries**
 - A type error is caused by a discrepancy between differing data types
- Type-safety is ensured by means of **metamodel objects**
 - queries are “assembled” programmatically from typed parts
 - more robust, because the Java compiler **can perform type-checking at compile-time**

Criteria API:

Canonical metamodel (1/2)

- Type checking requires a description of the objects in the application domain
- The **metamodel** is a set of objects that describe your domain model
 - The metamodel of a persistence unit is a description of the persistent type, state, and relationships of entities
- A **static metamodel** is a series of classes that mirror the entities in the domain model
 - They provide static access to the **metadata** about the mirrored class's attributes
 - In JPA terminology, the static metamodel is called **canonical metamodel**

Criteria API:

Canonical metamodel (2/2)

The canonical metamodel class for `Employee`

```
@StaticMetamodel(Employee.class)
public class Employee_ {
    public static volatile SingularAttribute<Employee, Integer> id;
    public static volatile SingularAttribute<Employee, String> name;
    public static volatile SingularAttribute<Employee, String> salary;
    public static volatile SingularAttribute<Employee, Department> dept;
    public static volatile SingularAttribute<Employee, Address> address;
    public static volatile CollectionAttribute<Employee, Project> project;
    public static volatile MapAttribute<Employee, String, Phone> phones;
}
```

- For each entity `E` in package `p`, a metamodel class `E_` in package `p` is created
- The metamodel class `E_` is annotated with the `@StaticMetamodel` annotation

Criteria API:

Building the query

- The essential steps in creating a Criteria API-based query are:
 - Creating an instance of the `CriteriaBuilder` class
 - Using this instance to create an instance of a `CriteriaQuery` class containing a query
 - Executing the query
- Where:
 - The `CriteriaQuery` instance represents the query we would like to execute
 - The `CriteriaBuilder` is a **factory** for all the individual pieces of the query

Criteria API:


Path expression

- Every `CriteriaQuery` defines at least one `Root` object, whose role is analogous to that of an identifier in a JPQL query
 - the `Root` object will form the basis for **path expressions** in the rest of the query

Path Expression (Criteria API)

```
emp.get(Employee_.department).get(Department_.id)
```

Equivalent to the JPQL expression
`emp.department.id`



- NB: using the canonical metamodel **enforces** type-safety

Criteria API:


Conditional expressions

- All of the conditional expression keywords, operators, and functions from JPQL are represented in some manner with the `CriteriaBuilder` interface

Finding the employee (Criteria API)

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> c =
    cb.createQuery(Employee.class);

Root<Employee> emp = c.from(Employee.class);
c.select(emp);
c.where(cb.equal(emp.get(Employee_.name), "John Smith"));
```



```
SELECT emp
FROM Employee emp
WHERE emp.name =
    "John Smith"
```



Criteria API:

JOIN operation

- Join can be performed by calling the `join()` method on the `Root` object
 - The resulting `Join` object behaves like `Root`, meaning that joins can be cascaded

Obtaining the John Smith's phone numbers

```
Root<Employee> emp = c.from(Employee.class);  
Join<Employee, Phone> j = emp.join(Employee_.phones);  
c.select(j.get(Phone_.number))  
c.where(cb.equal(emp.get(Employee_.name), "John Smith"));
```



```
SELECT p.number  
FROM Employee emp  
JOIN Phones p  
WHERE emp.name =  
"John Smith"
```

Criteria API:

WHERE clause

- Each condition in the WHERE clause is expressed as a `Predicate` instance
 - `Predicate`s can be combined by means of AND or OR operators

Conjunction of predicates (incremental construction)

```
Predicate predicates = cb.conjunction();
predicates = cb.and(predicates, cb.equal(emp.get(Employee_.name), "John Smith"));
predicates = cb.and(predicates, cb.equal(emp.get(Employee_.dept).get(Department_.id), 12));
c.where(predicates);
```

```
WHERE emp.name =
      "John Smith"
AND
emp.dept.id
= 12
```

Conjunction of predicates (list-based construction)

```
List<Predicate> predicates = new ArrayList<Predicate>();
predicates.add(cb.equal(emp.get(Employee_.name), "John Smith"));
predicates.add(cb.equal(emp.get(Employee_.dept).get(Department_.id), 12));
Predicate[] predArray = predicates.toArray(new Predicate[predicates.size()]);
c.where(cb.and(predArray));
```


Criteria API:

Defining parameters

- In JPQL is possible to define positional and named parameters
 - Parameters are just string aliases, e.g., `:name`
- To create a parameter using the Criteria API, a `ParameterExpression` object must be created

Defining a new query parameter

```
ParameterExpression<String> deptName = cb.parameter(String.class,  
"deptName");
```



Equivalent to the JPQL
parameter `:deptName`

Criteria API:

Executing typed queries

- JPA provides the `Query` interface to configure and execute queries.
 - The `Query` interface is used in cases when the result type is `Object`
- `TypedQuery` is a subinterface of `Query`
 - the `TypedQuery` interface is used in the typical case when typed results are preferred

Executing a Criteria API-based query

```
TypedQuery<Employee> q = em.createQuery(c);  
q.setParameter("deptName", "DEIB")  
return q.getResultList();
```

JPQL vs Criteria API

JPQL

```
@NamedQuery(  
    name="findProfByName",  
    query="SELECT p FROM Professor p  
    WHERE p.name LIKE :pName"  
)  
..  
res =  
em.createNamedQuery("findProfByN  
ame")  
.setParameter("pName", "Ceri")  
.getResultList();  
..
```

Criteria API

```
public List<Professor> findByName(String  
name) {  
    CriteriaBuilder cb = em.getCriteriaBuilder();  
    CriteriaQuery<Professor> q =  
        cb.createQuery(Professor.class);  
    Root<Professor> prof =  
        q.from(Professor.class);  
    q.select(prof);  
    ParameterExpression<String> nameP =  
        cb.parameter(String.class, "nameP");  
  
    q.where(cb.equal(prof.get("name"), nameP));  
  
    return q.setParameter("nameP", name)  
        .setMaxResults(10).getResultList();  
}
```

JPA IN WEB APPLICATIONS

Problem: Application layering

- Most Java Web Applications use some kind of application framework (e.g., Struts)
 - The objective is to separate presentation code, business-logic code and data-access code
- Mixing data-access code with application logic violates the emphasis on separation of concerns
 - Example: placing data-access code in a servlet or in a Struts Action
- JPA-related code should be hidden behind a façade, the so-called **persistence layer**

Data Access Object: design pattern

- The **Data Access Object** (DAO) design pattern encapsulates data-access logic, so as to increase reusability and maintainability
- The DAO design patterns allows developers to:
 - Manages the connection to the data source
 - Retrieve data
 - Update data
- There should be one DAO object for each entity so that all data operations related to the entity will be put inside the DAO itself

Data Access Object: code

EmployeeDAO.java (method signatures only)

```
public class EmployeeDAO {  
    public Employee findEmployee(int id) {...}  
    public List<Employee> findAllEmployees() {...}  
    public void removeEmployee(int id) {...}  
  
    public Employee raiseEmployeeSalary(int id, long raise) {...}  
}
```

Using a Data Access Object

Business-logic component (e.g., Struts' `Action` class)

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response){
    // Retrieve all the books
    BookDAO dao = new BookDAO();
    List<Book> books = dao.findAll();

    // Save the result set
    request.setAttribute("books", books);

    // Forward to the view
    return mapping.findForward("booklist");
}
```

Problem: Persistence context propagation

- In using JPA in a JSE environment, developers have to decide how to manage the `EntityManager`
- We do not want each DAO to **open, flush and close its own** persistence context
 - This is an anti-pattern known as **session-per-operation**
- One persistence context should support the whole **unit of work** (which may require several operations), not only one particular operation

Session per request pattern

- The most common pattern is known as **session-per-request**
 - A request from the client is sent to the server (where the persistence layer runs)
 - A new `EntityManager` is opened, and all DB operations are executed
 - Once the work has been completed (and the response has been prepared) the persistence context is flushed and closed, as well as the `EntityManager` instance
- A single database transaction is used to serve the client's request
 - The relationship between request and transaction is one-to-one

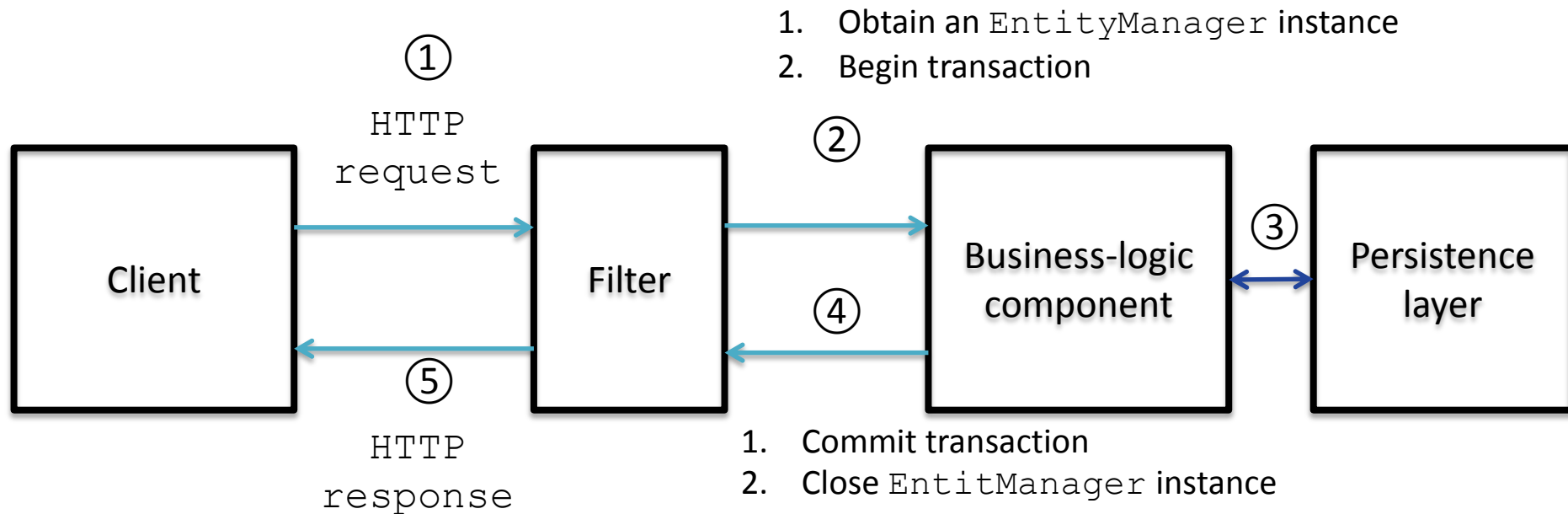
Persistence context propagation and transaction

- in a **JEE Environment**, session-per-request is the default persistence model
 - `EntityManager`s (injected or looked up) share the same persistence context for a whole transaction
- In a **JSE Environment**, the behavior has to be implemented by developers
 - The same `EntityManager` needs to be accessible to all DAO objects involved in the unit of work
 - `EntityManager` and transactions need to be started and ended correctly

Intercepting Filter design pattern (1/2)

- **Intercepting Filter** is a design pattern that intercepts and manipulates a request and a response before and after the request is processed
 - The transaction demarcation is implemented using an interceptor that runs when a request hits the servlet container
- In a servlet container this can be achieved by implementing the `ServletFilter` interface

Intercepting Filter design pattern (2/2)



Problem: Thread-safety (1/2)

- According to the session-per-request pattern, each user is associated with an `EntityManager` instance
 - `EntityManager` in JPA acts as a **session** object for a user
- Web tier components are meant to be used by multiple **concurrent** thread
 - The servlet container allocates a **new thread** for each new client's request
 - Servlet-based components are designed this way because they are intended to achieve high throughput through statelessness

Problem: Thread-safety (2/2)

- A class is **thread-safe** if it behaves correctly when accessed from multiple threads
 - `EntityManager` **is not** thread-safe, hence you should never inject it directly into a web component
- A `EntityManagerFactory` instead is a thread-safe, yet expensive-to-create, object intended to be shared by all application threads
 - It is created once, usually on application startup

ThreadLocal Session design pattern (1/2)

- `java.lang.ThreadLocal` class provides support for **thread-local variables**
 - Each thread that accesses a thread-local variable has its own, independently initialized copy of the variable
- The `EntityManager` can be bound to the thread that serves the request by using a `ThreadLocal` variable
 - The binding occurs in a helper class
 - This design pattern is called **ThreadLocal Session**

ThreadLocal Session design pattern (2/2)

EntityManagerHelper.java (ThreadLocal Session)

```
public class EntityManagerHelper {  
    private static final EntityManagerFactory emf;  
    private static final ThreadLocal<EntityManager> threadLocal;  
  
    static {  
        emf = Persistence.createEntityManagerFactory("BookStoreUnit");  
        threadLocal = new ThreadLocal<EntityManager>();  
    }  
  
    public static EntityManager getEntityManager() {  
        EntityManager em = threadLocal.get();  
        if (em == null) {  
            em = emf.createEntityManager();  
            threadLocal.set(em);  
        }  
        return em;  
    }  
    ...  
}
```

Each thread gets its own copy of the em variable

There exists only one EntityManagerFactory instance

Topics not covered

- Physical annotations
- Cascading policies other than `PERSIST`
- JPA in a JEE Environment
- Vendor-based APIs (e.g., Hibernate APIs)
- Advanced JPQL and Criteria API-based queries
- Long unit of works
- Connection pooling (e.g., C3P0)

References

- JPA 2.0 specification,
<http://www.jcp.org/en/jsr/detail?id=317>
- Hibernate Community Documentation,
EntityManager http://docs.jboss.org/hibernate/entitymanager/3.6/reference/en/html_single/
- Hibernate Community Documentation,
Transactions and Concurrency
<http://docs.jboss.org/hibernate/orm/4.0/hem/en-US/html/transactions.html>

References

- Hibernate Wiki, Data Access Objects
<https://community.jboss.org/wiki/GenericDataAccessObjects>
- Hibernate Wiki, *Open Session in View pattern*
<https://community.jboss.org/wiki/OpenSessionInView>
- Hibernate Community Documentation, *Metamodel*
<http://docs.jboss.org/hibernate/orm/4.0/hem/en-US/html/metamodel.html>
- Hibernate Community Documentation, *Criteria Queries*
<http://docs.jboss.org/hibernate/entitymanager/3.5/reference/en/html/querycriteria.html>

References

- Pro JPA 2, Mastering The Java Persistence API, *M. Keith, M. Schincariol, Apress Media LLC*
- EJB3 In Action, *D. Panda, R. Rahman, D. Lane, Manning Publications Co.*
- Java Persistence With Hibernate, *C. Bayer, G.King, Manning Publications Co.*
- Hibernate Recipes, A Problem-Solution Approach, *S. Guruzu G. Mak, Apress Media LLC*
- Enterprise JavaBeans 3.1, *A.L. Rubinger, B.Burke, O'Reilly Media*
- EJB 3.1 Cookbook, *R.M. Reese, Packt Publishing*