# Open Source Frameworks (OSF)
# Persistence with JPA

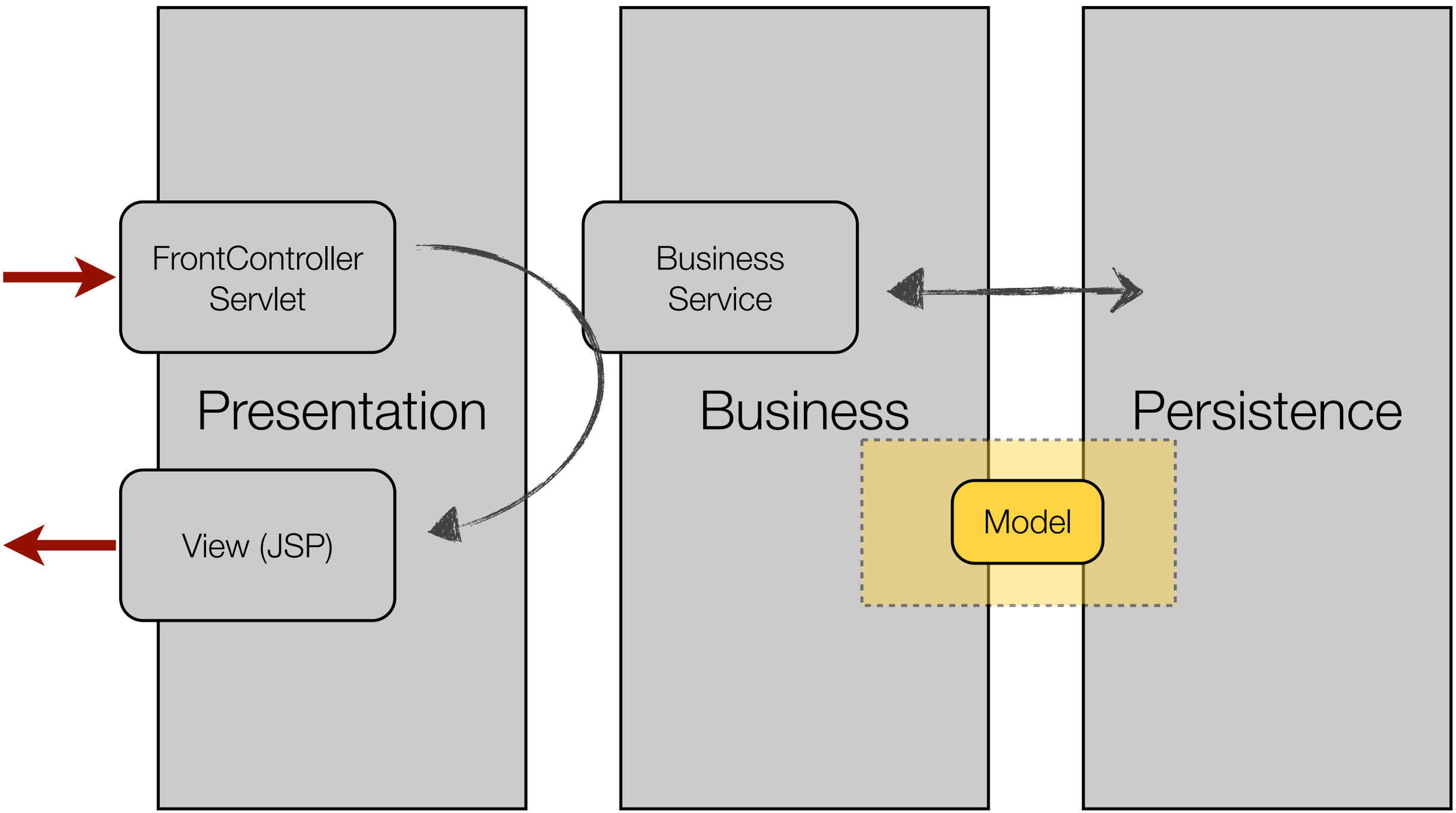
Open Source Frameworks (OSF)
Master of Science in Engineering (MSE)
Olivier Liechti
olivier.liechti@heig-vd.ch

MSE | MASTER OF SCIENCE IN ENGINEERING

# Planning

| Date | Java EE Frameworks | Gamification Project |
|---|---|---|
| 23.09.13 | Intro, Java EE Overview, EJBs | Environment setup 1 |
| 30.09.13 | REST APIs & JAX-RS | Environment setup 2 (automation) |
| 07.10.13 | Design and document a REST API for your gamification engine | |
| 14.10.13 | Persistence with JPA | Test and implement your REST API |
| 21.10.13 | Break | |
| 28.10.13 | Test and implement your REST API | |
| 04.11.13 | Spring Framework | **Presentations & demos** |
| 11.11.13 | Technical POC Project: Define the scope & plan the activities | |
| 18.11.13 | Technical POC Project: Build the reference system | |
| 25.11.13 | Technical POC Project: Build the test infrastructure | |
| 02.12.13 | **Technical POC Project: Present the results (with a demo)** | |
| 09.12.13 | Introduction to Javascript frameworks | Get ready with node.js & express |
| 16.12.13 | Re-implement your REST API in Javascript | |
| 23.12.13 | Break | |
| 30.12.13 | | |
| 06.01.14 | Re-implement your REST API in Javascript | |
| 13.01.14 | Java Message Service | **Presentations & demos** |

# The Business Tier



# Agenda

- Approaches to persistence
  - “Direct” approach (e.g. JDBC)
  - Object-Relational Mapping (ORM) approach
- Java Persistence API
  - Persistence as a service provided by the environment
  - Programming model and abstractions defined in the API
- References
  - Java EE tutorial
  - http://weblogs.java.net/blog/2006/06/09/ejb-30-sessions-2006-javaone
  - http://www.agiledata.org/essays/impedanceMismatch.html

# Approaches to persistence

- “Traditional” approach
  - JDBC is a standardized API for interacting with a RDBS from Java.
  - You connect to the DB, submit SQL requests, process tabular result sets
  - In the J2EE days, the Data Access Object (DAO) was very popular to create a persistence layer in your application (e.g. CustomerDAO)
- Object Relational Mapping (ORM) approach
  - You use a higher-level API and don’t directly work at the SQL level
  - You declaratively specify how your object-oriented model should be mapped to a relational model
  - The middleware takes care of the SQL queries.
- The two approaches are **not mutually exclusive**: even if you use an ORM, it is sometimes useful/necessary to have low-level access to SQL. **Balance** productivity with performance.

# Java Persistence API

- Java Persistence API is an Object Relational Mapping (ORM) API
- Java Persistence API is defined in JSR 220 (JPA 2.0 in 317, JPA 2.1 in 338)
- **Step 1: you design your object-oriented domain model**
  - With JPA, every business object is defined as an “entity”
  - Some coding conventions are defined for JPA entities
  - The persistence properties and behavior are specified declaratively with special annotations (XML is also possible)
- **Step 2: you interact with a “persistence service”**
  - The environment provides a “persistence service”, that one can use to find, insert, update and delete business objects
  - JPA defines interfaces and classes for this “persistence service”
  - Note: JPA can be used in the EJB container, in the Web container, but also in Java SE applications!

# Java Persistence API

*With JPA, you define an object-oriented domain model. You work with business objects, specify relationships between them.*

***You live in the wonderful world of objects.***

*And you let JPA handle the interactions with the database. The schema can be generated automatically, the SQL queries as well.*

# Java Persistence API

*With JPA, like with other Java EE API, you can rely on* ***conventions****. You don't have to explicitly specify all aspects of the configuration. If you don't, the* ***standard behavior*** *applies.*

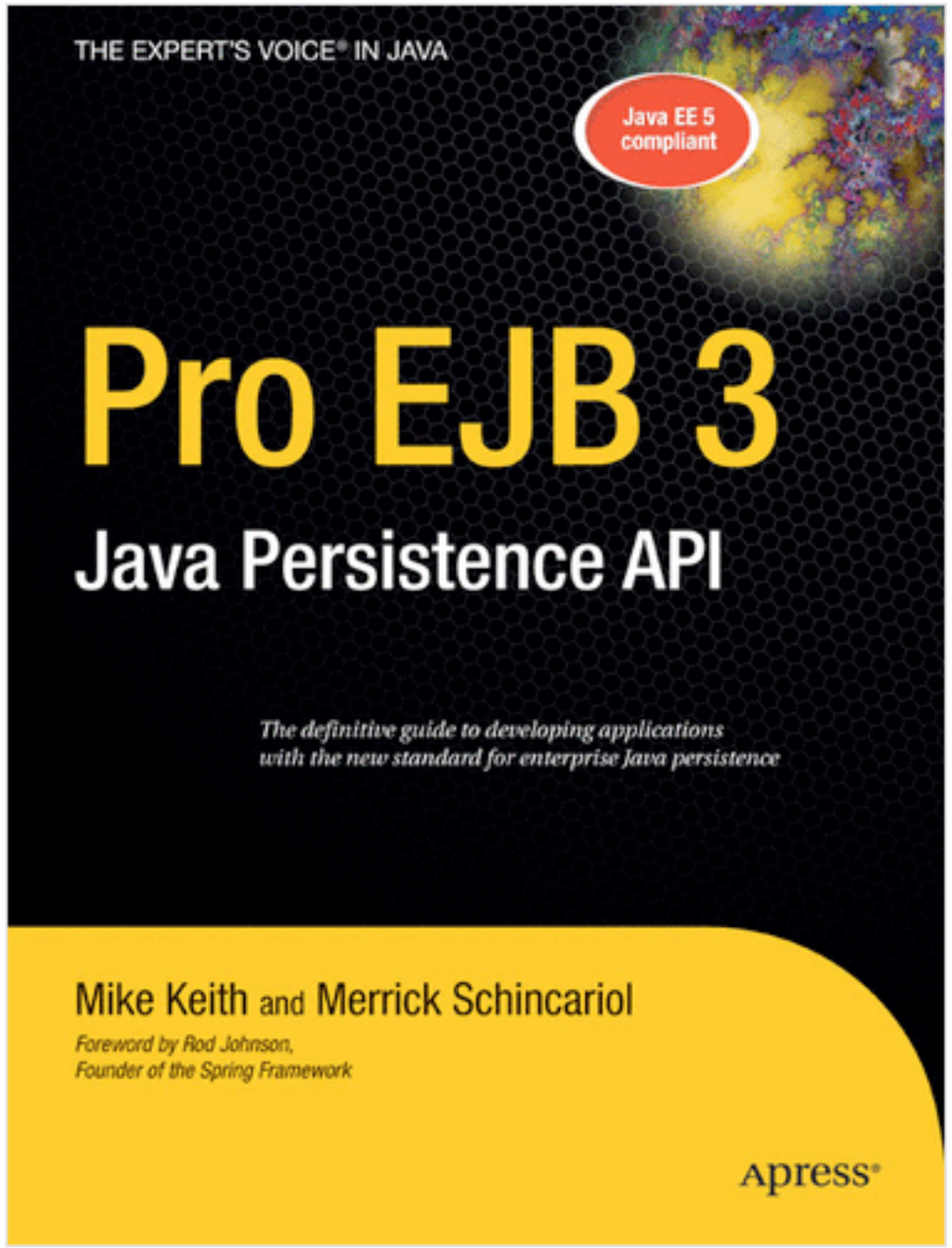
*But you* ***stay in control****: if there is something that you don't like about the default behavior, you can change it with different annotations.*

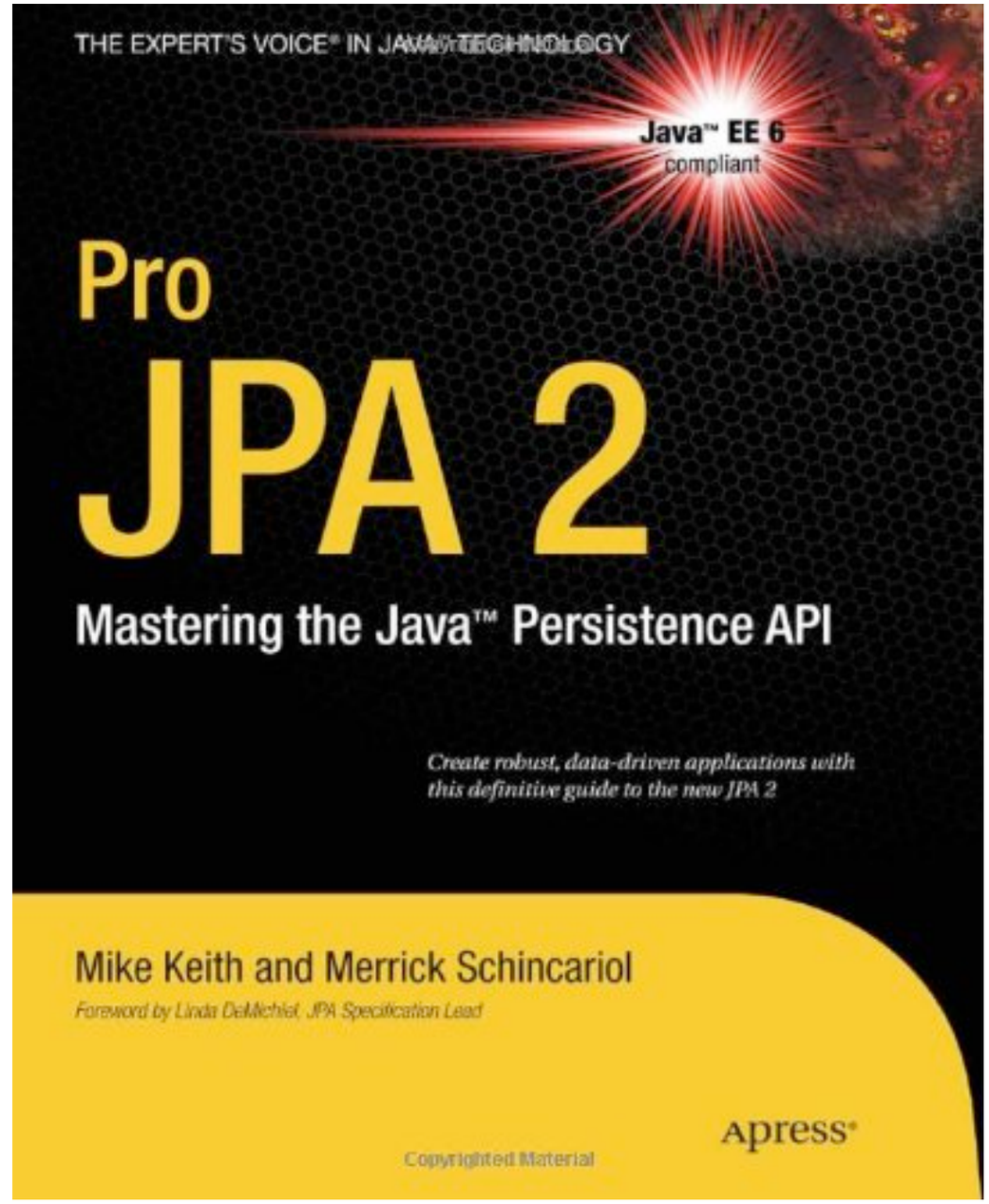
# Java Persistence API

*If you start a project from scratch and do not have to use an existing database, you can* ***generate the schema*** *from the Java model. In general, specifying the OR mapping will be pretty easy...*

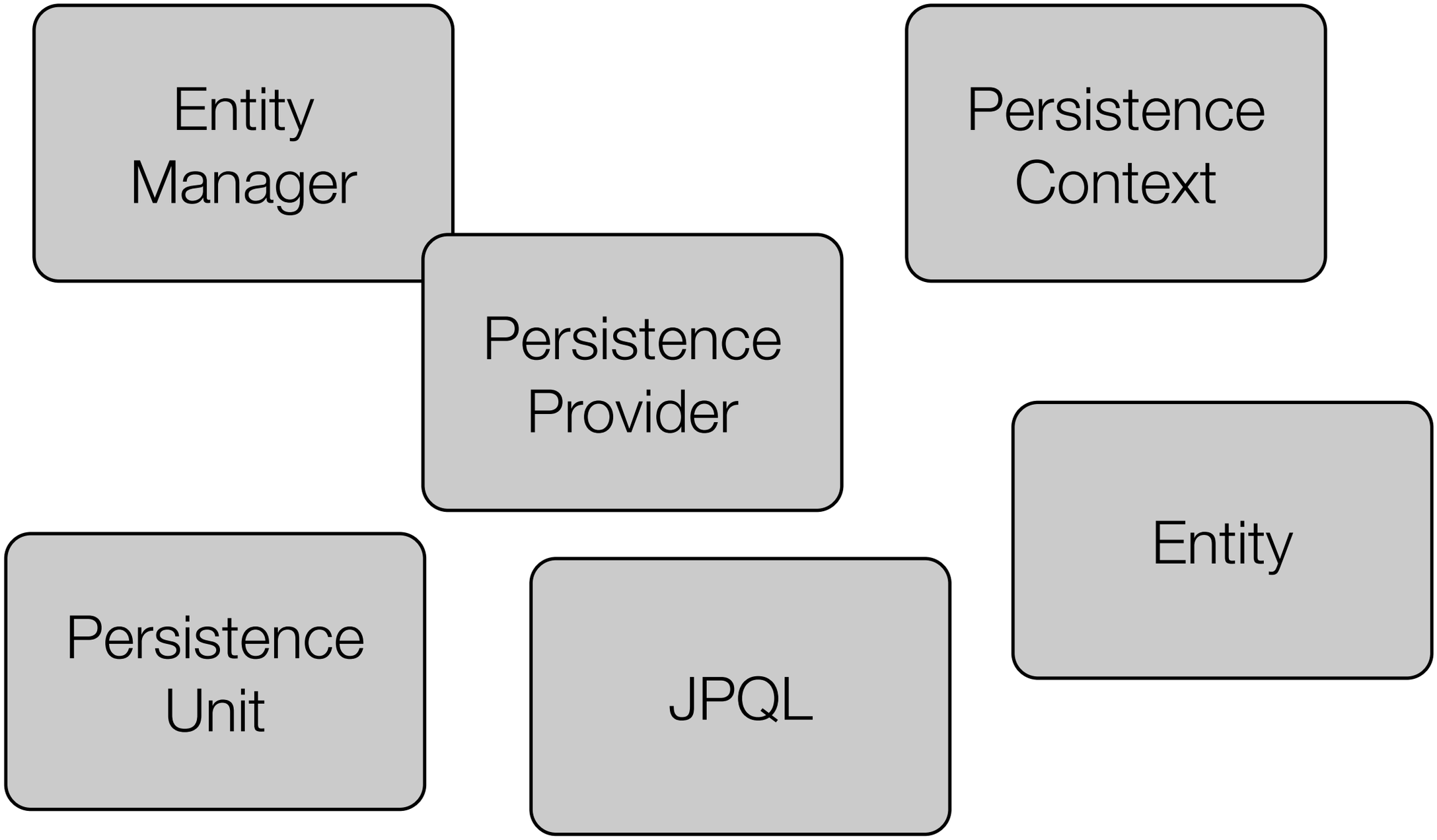
*If you have an* ***existing database schema****, then you will need fine control over the OR mapping. JPA gives you this control.*

# The JPA Bible



# Abstractions defined in the JPA API



# Persistence Provider

- A Persistence Provider is an **implementation** of the JPA API.
- **TopLink Essentials** and **Hibernate** are two examples of JPA Persistence Providers.
- Persistence Providers are “pluggable”. This means that if you use only standard JPA features, you can for example decide to switch from TopLink to Hibernate at some point.
- Many JPA Persistence Providers have been created on the basis of **existing ORM solutions** (Hibernate existed before JPA, TopLink as well).
- Many Persistence Providers give you access to non-standard features. **Balance** functionality with portability...

# JPA entities

- **Remember**: it is not the same thing as a J2EE 1.x/2.x Entity Bean (EJB).
- It is a Plain Old Java Object (**POJO**).
- It does not need to extend any particular class, nor to implement any particular interface.
- This is important, because inheritance can be used to capture business domain relationships (vs. for technical reasons).
- It has a "**persistent state**", i.e. a set of attributes that should be saved in the persistent store.
- An entity can have **relationships** with other entities. **Cardinality** and **navigability** can be specified for every relationship.

```java
@Entity
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String firstName;
    private String lastName;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
...
}
```

This is an entity class

An entity needs a unique id

There are different ways to generate these id values

The attributes will be automatically part of the "persistent state" for this entity.

*If you do not want to persist a field, use the @Transient annotation*

# Requirement for a JPA Entity

- The class must be annotated with the `javax.persistence.Entity` annotation.
- The class must have a **public or protected, no-argument constructor**. The class may have other constructors.
- The class must not be declared final. No methods or persistent instance variables must be declared final.
- If an entity instance be passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the **Serializable** interface.
- Entities may **extend** both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class's methods. **Clients must access the entity's state through accessor or business methods**.

# Entity Relationships

- Cardinalities
  - one-to-one
  - one-to-many
  - many-to-many
  - many-to-one
- Bi-directional relationships
  - **Warning**: the developer is responsible for maintaining both “sides” of the relationship!

```
employee.setOffice(office);
office.setEmployee(employee);
```

- Key questions
  - loading behavior: eager vs. lazy
  - cascading behavior: cascading or not? for what operations?

# Entity Relationships

```
@Entity public class Customer {
    @Id protected Long id;
    …
    @OneToMany protected Set<Order> orders = new HashSet();
    @ManyToOne protected SalesRep rep;
    …
    public Set<Order> getOrders() {return orders;}
    public SalesRep getSalesRep() {return rep;}
    public void setSalesRep(SalesRep rep) {this.rep = rep;}
}

@Entity public class SalesRep {
    @Id protected Long id;
    …
    @OneToMany(mappedBy="rep")
    protected Set<Customer> customers = new HashSet();
    …
    public Set<Customer> getCustomers() {return customers;}
    public void addCustomer(Customer customer) {
        getCustomers().add(customer);
        customer.setSalesRep(this);}
}
```

# Entity Relationships

```
@Entity
public class Customer {
   @Id
   int id;
   ...
   @ManyToMany
   Collection<Phone> phones;
}
```

```
@Entity
public class Phone {
   @Id
   int id;
   ...
   @ManyToMany(mappedBy="phones"
   Collection<Customer> custs;
}
```

# Entity Manager

- The Entity Manager is the **interface** to the "**persistence service**".
- In other words, it is through the Entity Manager that you:
  - **retrieve** and **load** information from the database
  - **create** new information in the database
  - **delete** data information the database

```
javax.persistence.EntityManager

<T> T find(Class<T> entityClass, Object primaryKey);
void persist(Object entity)
void remove(Object entity)
Query createNamedQuery(String name)
Query createNativeQuery(String sqlString)
...
```

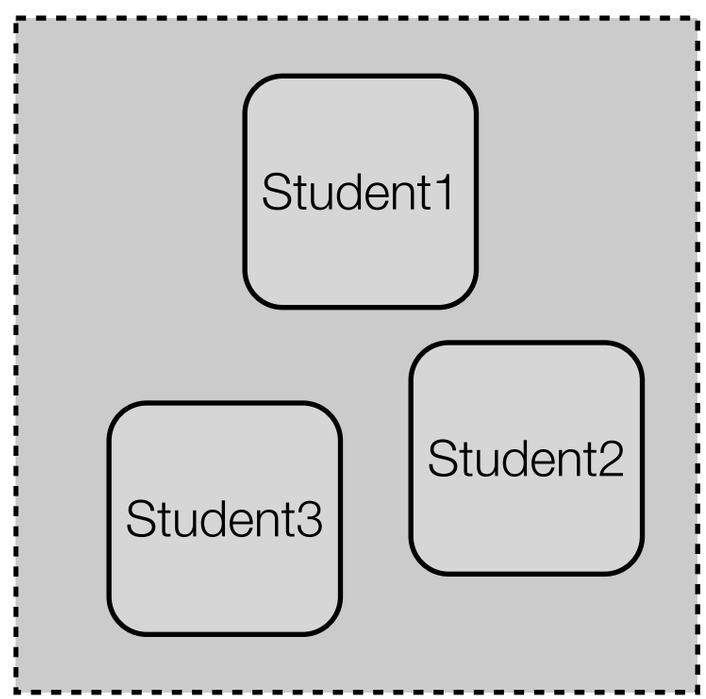
# Using the Entity Manager

- You can use the Entity Manager in **different types of components**: EJBs, servlets, java applications, etc.
- Using the Entity Manager from **EJBs** is easy. You simply ask the container to inject a reference to the Entity Manager in a variable, with an annotation.
- Using the Entity Manager in the **web tier** requires some care to deal with concurrency (`EntityManager` is not thread-safe, `EntityManagerFactory` is thread-safe).

```
@Stateless
public class StudentsManagerBean implements StudentsManagerLocal {
    @PersistenceContext
    EntityManager em;
    public long createStudent(String firstName, String lastName) {
        Student student = new Student();
        student.setFirstName(firstName); student.setLastName(lastName);
        em.persist(student); em.flush();
        return student.getId();
    }
}
```

# Persistence Context

- A Persistence Context is a set of entity instances at **runtime**.
- Think of a temporary “bag” of objects that come from the database, that are managed by JPA and that will go back to the database at some point.
  - If you modify the state of one of these objects, you don’t have to save it explicitly. It will be persisted back automatically at commit time.
- Using the JPA API, you can manage the persistence context, populate it, etc.

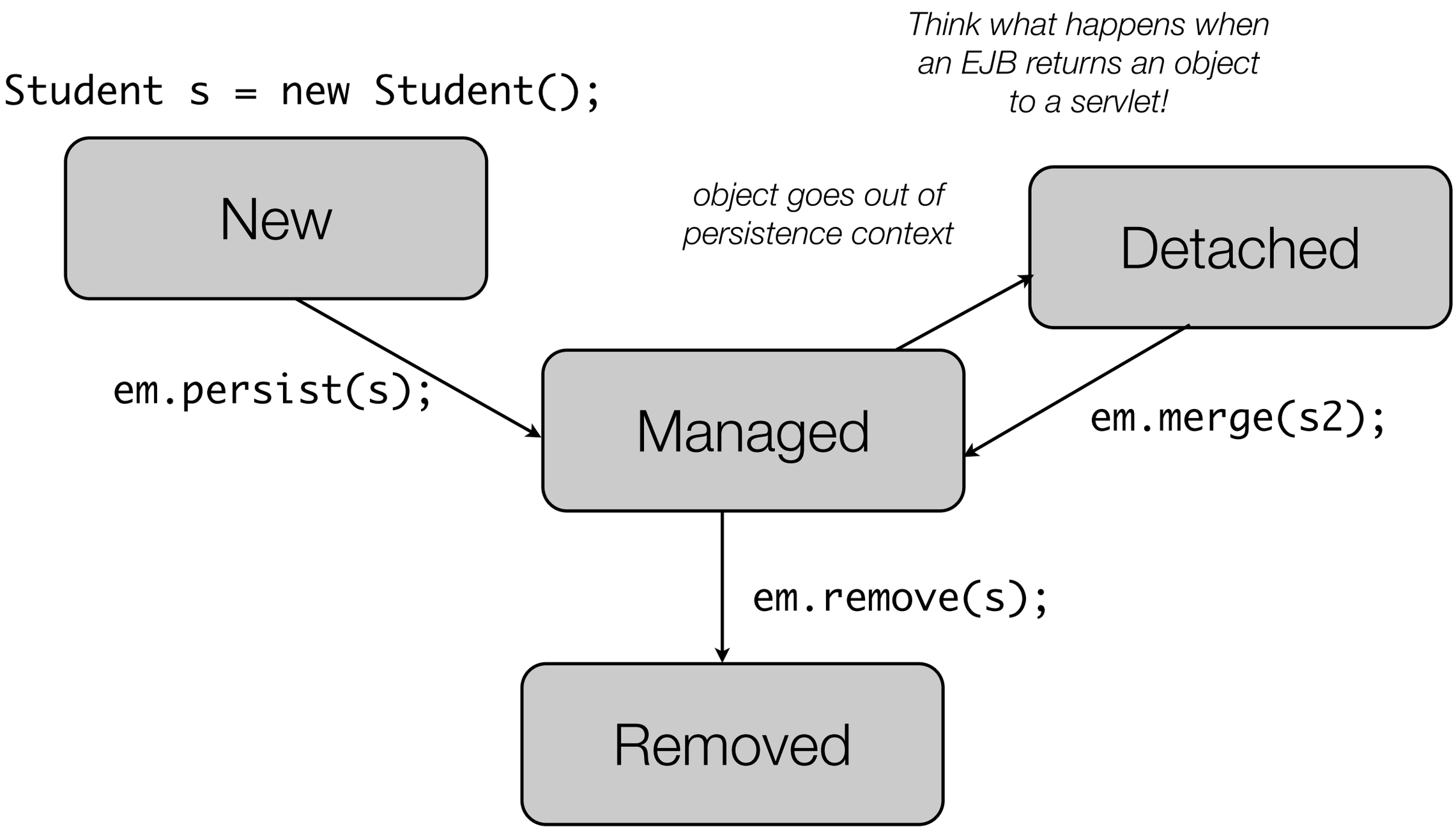



Persistence Context

# Persistence Context

- "A persistence context is a **set of managed entity instances** in which for any persistent entity identity there is a **unique entity instance**.
- Within the persistence context, the entity instances and their **lifecycle** are managed by the **entity manager**."
  - "A **new entity instance** has no persistent identity, and is not yet associated with a persistence context.
  - A **managed entity instance** is an instance with a persistent identity that is currently associated with a persistence context.
  - A **detached entity instance** is an instance with a persistent identity that is not (or no longer) associated with a persistence context.
  - A **removed entity instance** is an instance with a persistent identity, associated with a persistence context, that is scheduled for removal from the database."

# Life-cycle for JPA Entities



# Persistence Context Types

- In Java EE, we typically use a **transaction-scoped persistence context**:
  - The client invokes a method on a **Stateless Session Bean**
  - The container intercepts a call and **starts a transaction**
  - The Stateless Session Bean uses JPA, a persistence context is created
  - Entities are loaded into the **persistence context**, modified, added, etc.
  - The method returns, the container **commits** the transaction
  - At this stage, entities in the persistence context are **sent back** to the DB.
- JPA also defines **extended persistence context**:
  - Entities remain managed as long as the Entity Manager lives
  - The JBoss SEAM framework uses extended persistence contexts: a persistence context lives during a whole “conversation”.

# Persistence Unit

- The Persistence Unit defines a list of entity classes that “belong together”.
- All entities in one Persistence Unit are stored in the same database.
- Persistence Units are declared in `persistence.xml` file, in the `META-INF` directory of your `.jar` file (it is possible to define several Persistence Units in the same xml file).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence ...>
  <persistence-unit name="SimplePersistenceExample-ejbPU" transaction-type="JTA">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <jta-data-source>jdbc/demo</jta-data-source>
    <class>ch.heigvd.osf.demo.model.Student</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

# Java Persistence Query Language (JPQL)

- SQL-like query language
- Includes constructs for exploiting the OR mapping. For instance, you can define polymorphic queries if you have defined inheritance relationships.

```
SELECT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

```
public List findWithName(String name) {
  return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

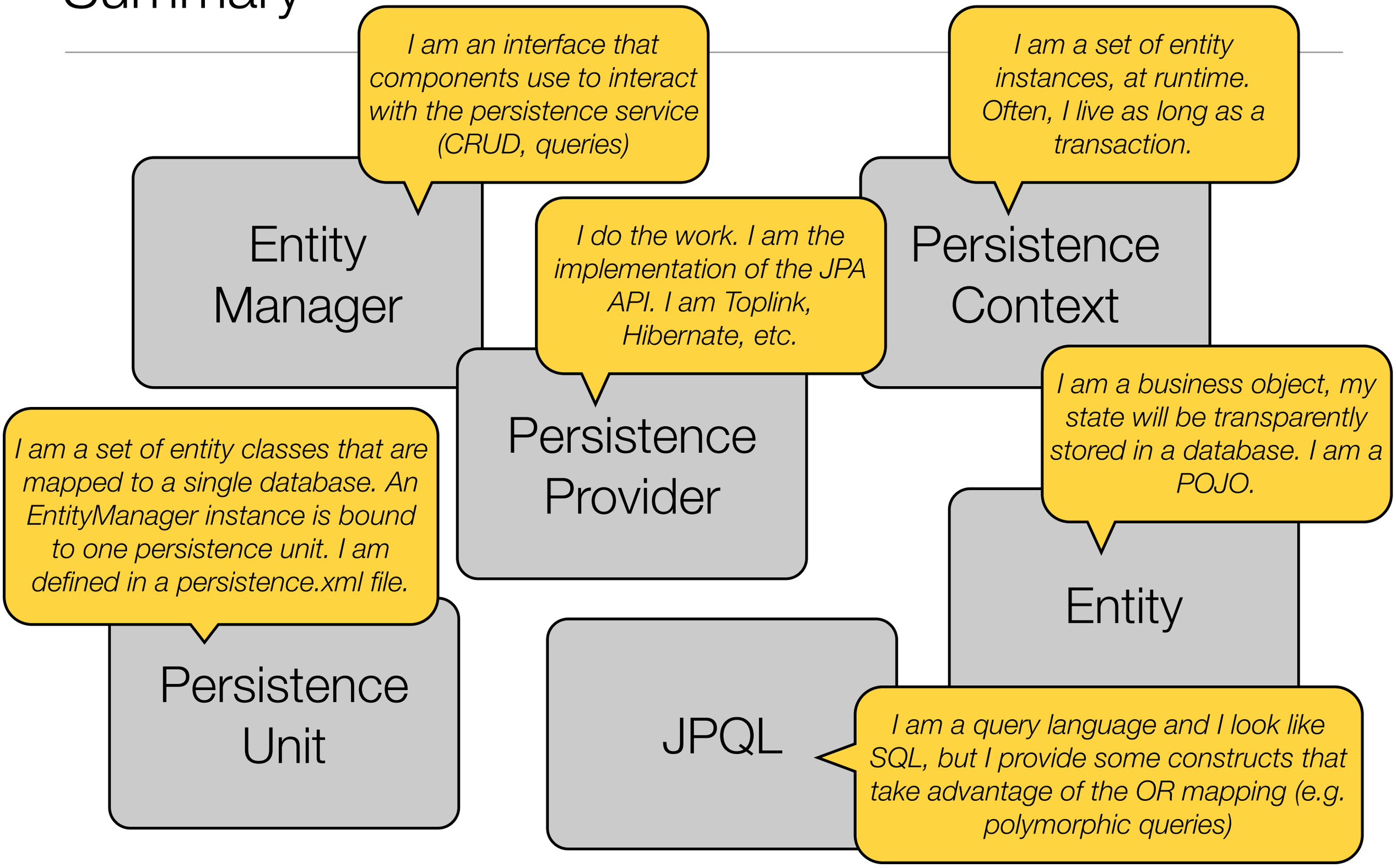
# Java Persistence Query Language (JPQL)

- You can group all your queries at the same place (vs. directly in the service method). Common practice is to use the @NamedQuery in the Entity Class source.

```
@NamedQuery(
name="findAllCustomersWithName",
query="SELECT c FROM Customer c WHERE c.name
LIKE :custName"
)
```

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
.setParameter("custName", "Smith")
.getResultList();
```

# Summary


I am an interface that components use to interact with the persistence service (CRUD, queries)
Entity Manager
I do the work. I am the implementation of the JPA API. I am Toplink, Hibernate, etc.
Persistence Provider
I am a set of entity instances, at runtime. Often, I live as long as a transaction.
Persistence Context
I am a set of entity classes that are mapped to a single database. An EntityManager instance is bound to one persistence unit. I am defined in a persistence.xml file.
Persistence Unit
I am a business object, my state will be transparently stored in a database. I am a POJO.
Entity
JPQL
I am a query language and I look like SQL, but I provide some constructs that take advantage of the OR mapping (e.g. polymorphic queries)

# Transactions

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- **Transactions**
  - ACID
  - Distributed transactions
- **Transactions in Java EE**
  - Container managed transactions
  - Transactions and annotations
  - Transactions and JMS

# Transactions

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

```
transaction.start();

    accountA.debit(100);
    accountB.credit(100);

    transaction.commit();
```

# Transactions



```
transaction.start();

accountA.debit(100);
try {
  accountB.credit(100);
} catch (AccountFullException e) {
  transaction.rollback();
}

transaction.commit();
```

# ACID

# ACID

Atomicity: “all or noting”

# ACID

Consistency: “business data integrity”

# ACID

Isolation: “deal with concurrent transactions”

# ACID

Durability: “once it’s done, it’s done”

# Transaction Demarcation

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

```
transaction.start();

accountA.debit(100);
accountB.credit(100);

transaction.commit();
```

**Who** does the “start” and the “commit” and **where**?

# Container Managed Transaction

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- The EJB container handles calls to `commit` and `rollback`.
- Methods defined on EJBs provide demarcation points.
- This is the **default behavior**.

```
<<Servlet>>
CustomerController
 ...
 cm.invoice(29, 2000);
 ...
```

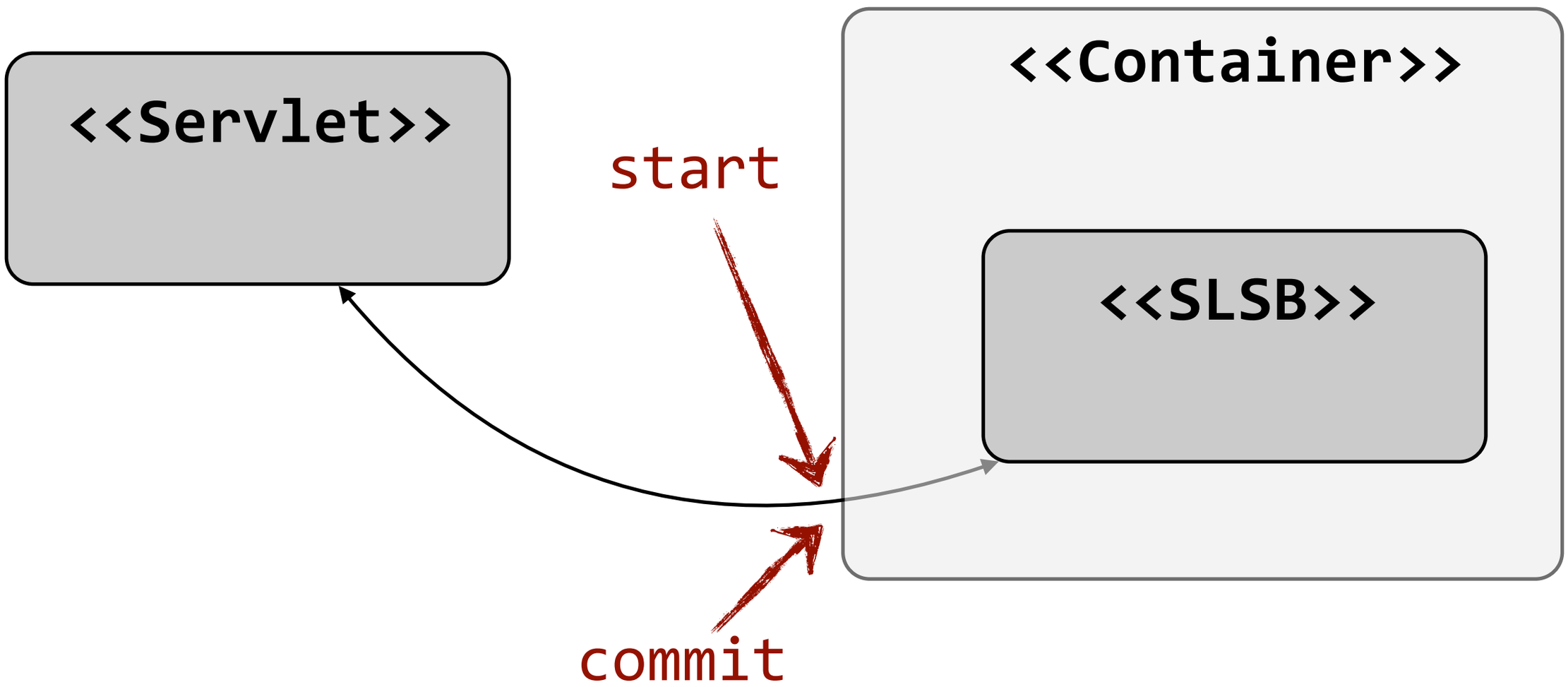
```
<<SLSB>>
CustomerManager

public invoice(
  long id, int amount);
```

# Container Managed Transaction

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

- The EJB container handles calls to `commit` and `rollback`.
- Methods defined on EJBs provide demarcation points.
- This is the **default behavior**.



# Transaction Scope



What happens when a **client** calls a method on a session bean, which calls a method on a session bean, which calls a method on a session bean, which calls a method on a session bean, which calls a method on a session bean, which calls a method on a session bean, which calls a method on a session bean, which calls a method on a session bean, which **throws an exception**?

# Transaction Scope



What h[illegible]hen a **client** calls a method on [illegible]calls a method on a [illegible]ls a method on a session bea[illegible] a method on a session bean, which calls a method on a se[illegible] which calls a method on a se[illegible] method on a sessio[illegible] **exception**?

**Everything** should be rolled back!

**No!** Only changes incurred by the last method should be rolled back!

# Transaction Scope

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

What h[illegible] when [illegible]

**Everything** should [illegible] rolled back!

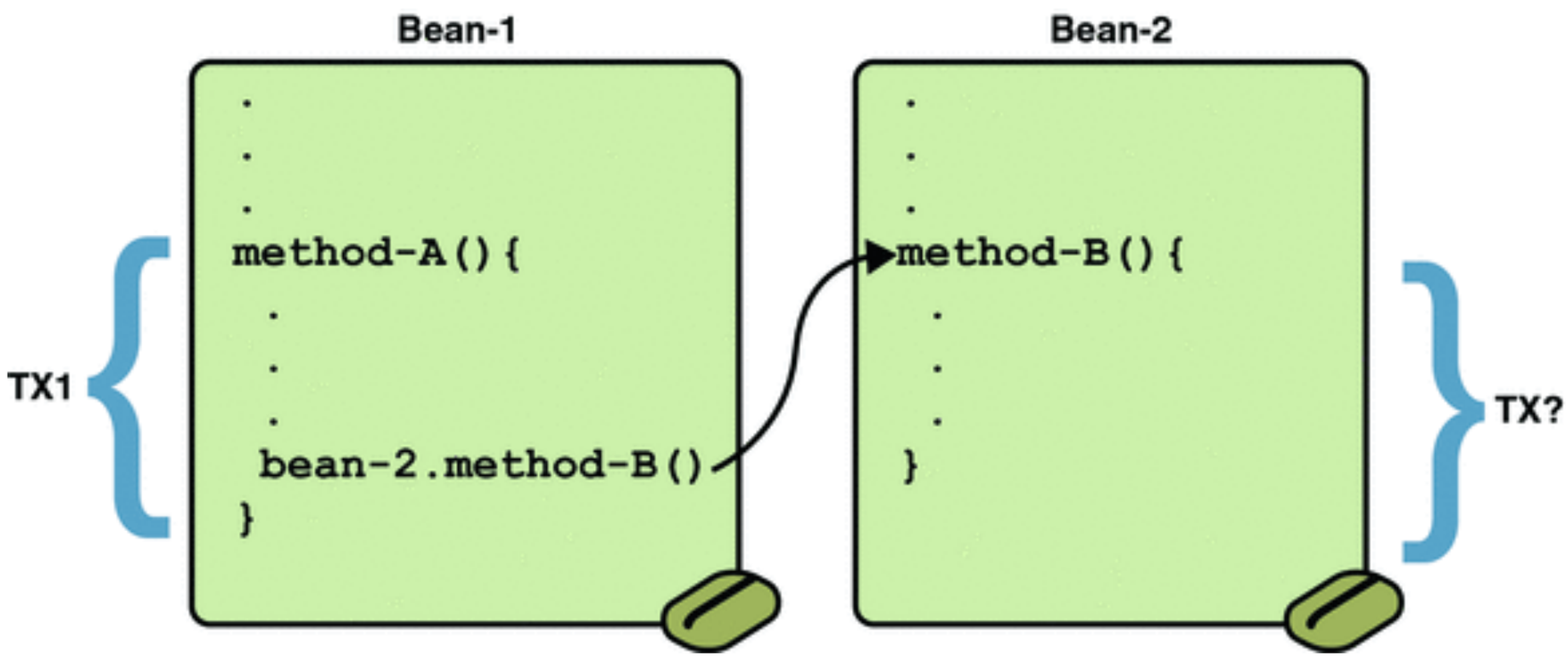
bea[illegible]
which calls a method [illegible]
calls a method on a se[illegible]
method on a session [illegible]
**exception**?

It is **up to the application** to specify intended behavior. The developer must specify transaction scope, typically with **annotations**.

**No!** Only changes incurred by the last method should be rolled back!

# Transaction Scope

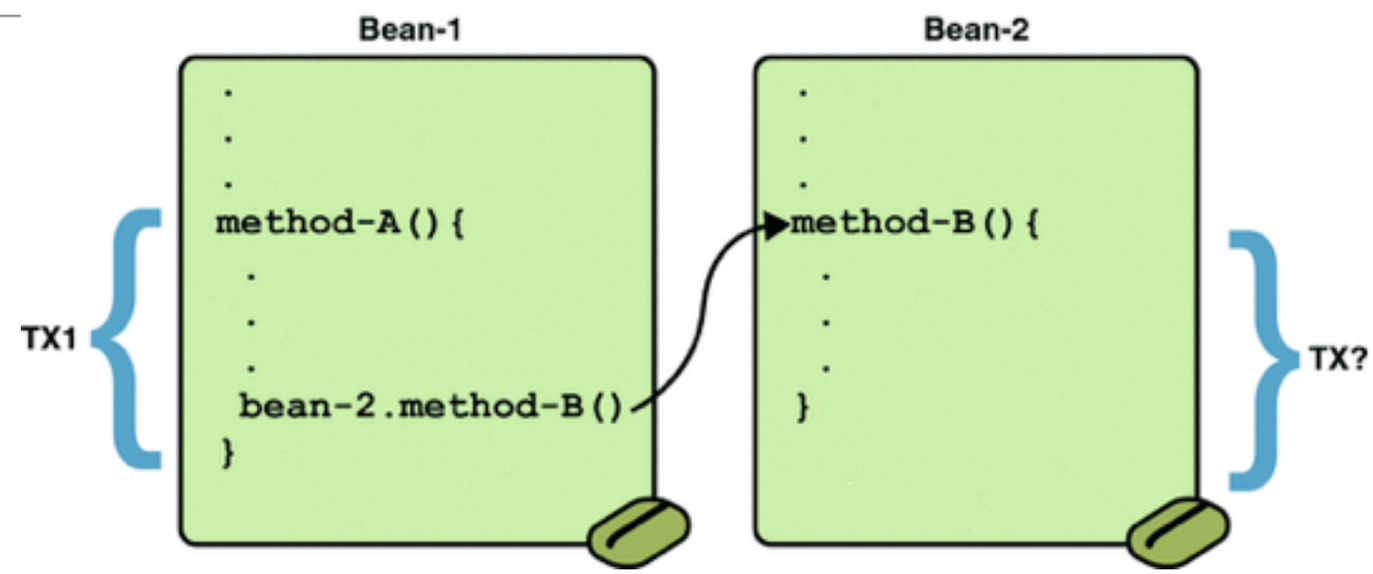
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud




http://java.sun.com/javaee/5/docs/tutorial/doc/bncij.html

# Transaction Scope

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud




```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements
Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

    public void fourthMethod() {...}
}
```

| Transaction Attribute | Client's Transaction | Business Method's Transaction |
|---|---|---|
| Required | None | T2 |
| | T1 | T1 |
| RequiresNew | None | T2 |
| | T1 | T2 |
| Mandatory | None | error |
| | T1 | T1 |
| NotSupported | None | None |
| | T1 | None |
| Supports | None | None |
| | T1 | T1 |
| Never | None | None |
| | T1 | Error |

# Transactions & Exceptions



There are two ways to roll back a container-managed transaction.

First, if a **system exception** is thrown, the container will automatically roll back the transaction.

Second, by invoking the **setRollbackOnly** method of the EJBContext interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the rollback is not automatic but can be initiated by a call to setRollbackOnly.

**Note**: you can also annotate your Exception class with `@ApplicationException(rollback=true)`

# Bean Managed Transactions

heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

**If you have special needs**, you can control the transaction demarcation yourself. You will start and commit the transactions, and possibly rollback them.

To do that, you need to use Bean Managed Transactions.

```
Stateful
@TransactionManagement(BEAN)
public CartSession {
  CartEnt cart;
  @PersistenceContext EntityManager em;
  @Resource UserTransaction ut;

  @PostConstruct public startCart() {
       ut.begin();
       cart = new CartEnt();
  }

  public addItem (String itemid, int qty) {
      em.persist(new CartItem(itemid, qty, cart.getId());
      cart.setItemQuantity(cart.getItemQuantity() + 1);
  }

  public checkOut() {
      em.merge(cart);
      ut.commit();
  }
}
```