

07 - Persistence tier (2)

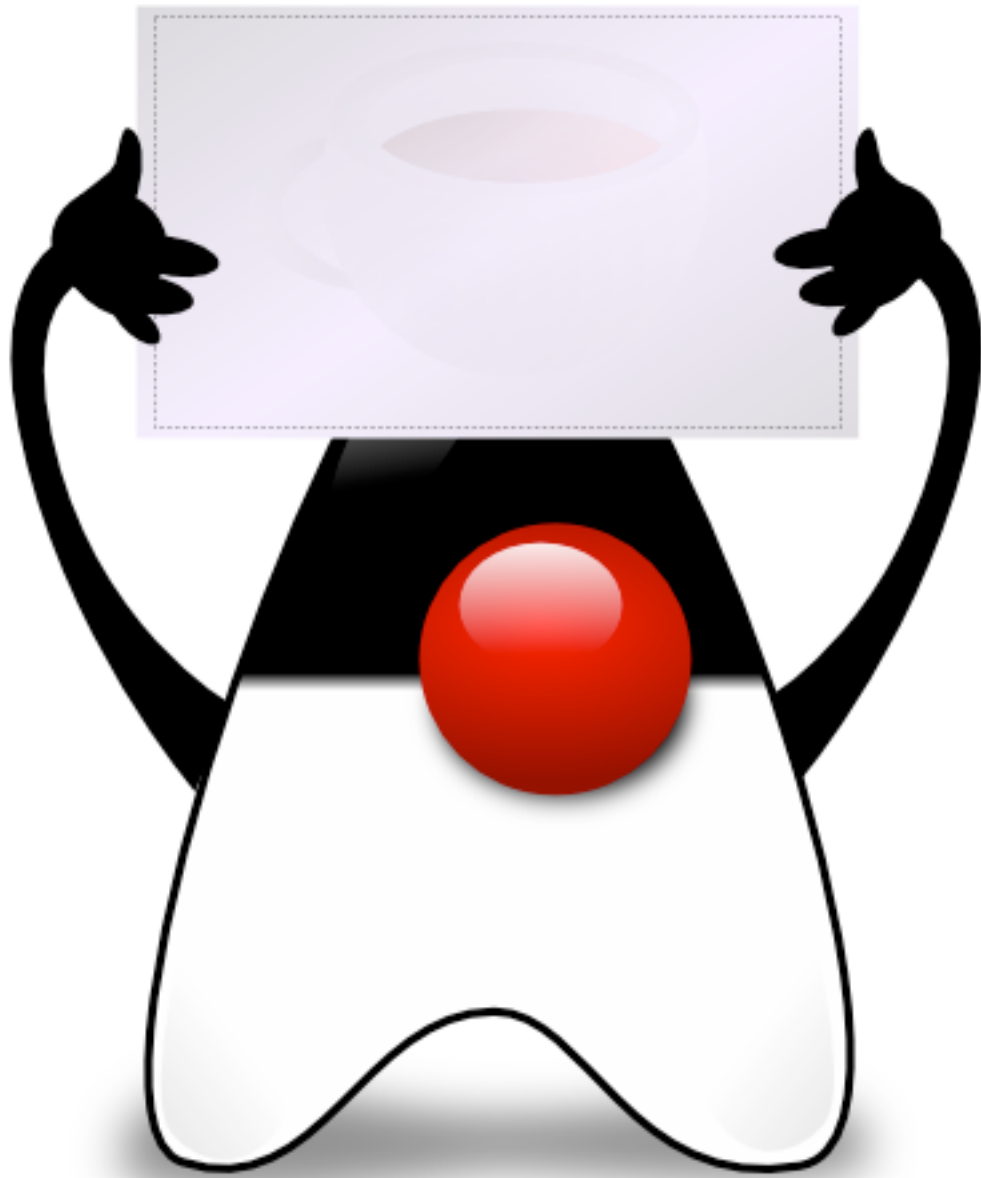
ORM: Java Persistence API (JPA)

AMT 2019

Olivier Liechti

Foreword


For the first project, we stick to plain old JDBC. You will have the time to put JPA in practice in the second project, with Spring Data JPA.



Transactions (with EJBs)

Transactions

Imagine that you have the following code in a business service:

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

What happens if the application crashes here?
Is my data corrupted?
Has money vanished in cyberspace?

transaction.start();

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

transaction.commit();

Transactions give us an “whole or nothing” semantic
(we often speak about a unit of work)

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

We can also deal with application-level errors and leave the data in a consistent state.

ACID

ACID

Atomicity: “all or nothing”

ACID

Consistency: “business data integrity”

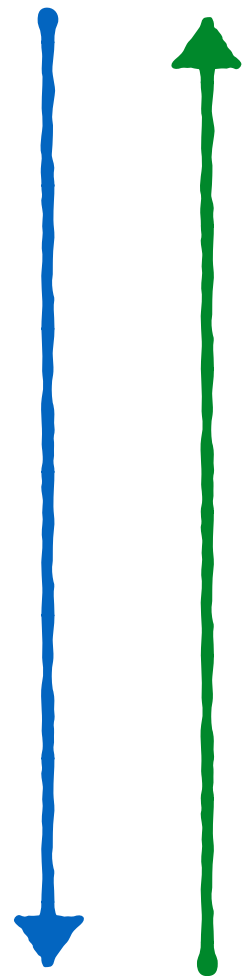
ACID

Isolation: “deal with concurrent transactions”

There are different isolation levels!

Isolation levels

Increasing isolation
between
transactions



Isolation level	Potential issues
Read Uncommitted (no locks)	Dirty Reads (no isolation)
Read Committed (write locks)	Non-repeatable Reads
Repeatable Reads (read & write locks)	Phantom reads
Serializable (range locks)	

Increasing performance in the
cas of concurrent access

Isolation levels

- "A **dirty read** occurs when a transaction is allowed to read data from a row that has been modified by another running transaction and not yet committed."
- "A **non-repeatable read** occurs, when during the course of a transaction, a row is retrieved twice and the values within the row differ between reads."
- "A **phantom read** occurs when, in the course of a transaction, two identical (SELECT) queries are executed, and the **collection** of rows returned by the second query is different from the first."

See for **example scenarios**, see:

https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html#transactions_data_integrity

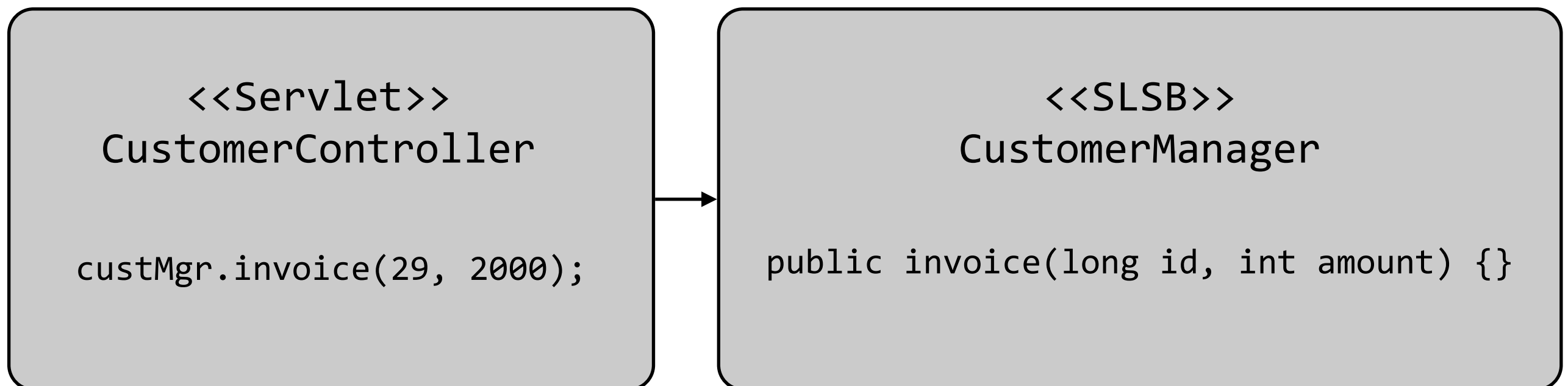
[https://en.wikipedia.org/wiki/Isolation_\(database_systems\)#Read_phenomena](https://en.wikipedia.org/wiki/Isolation_(database_systems)#Read_phenomena)

ACID

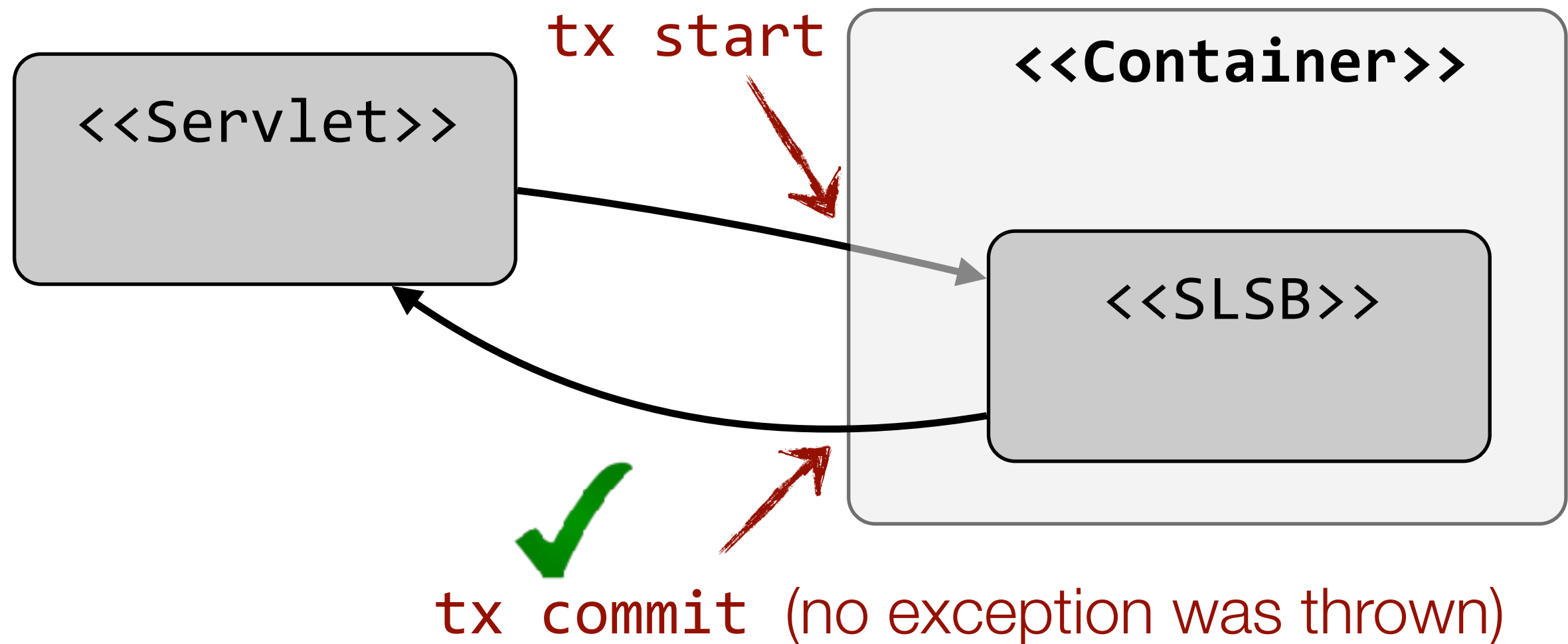
Durability: “once it’s done, it’s done”

Transactions & EJBs

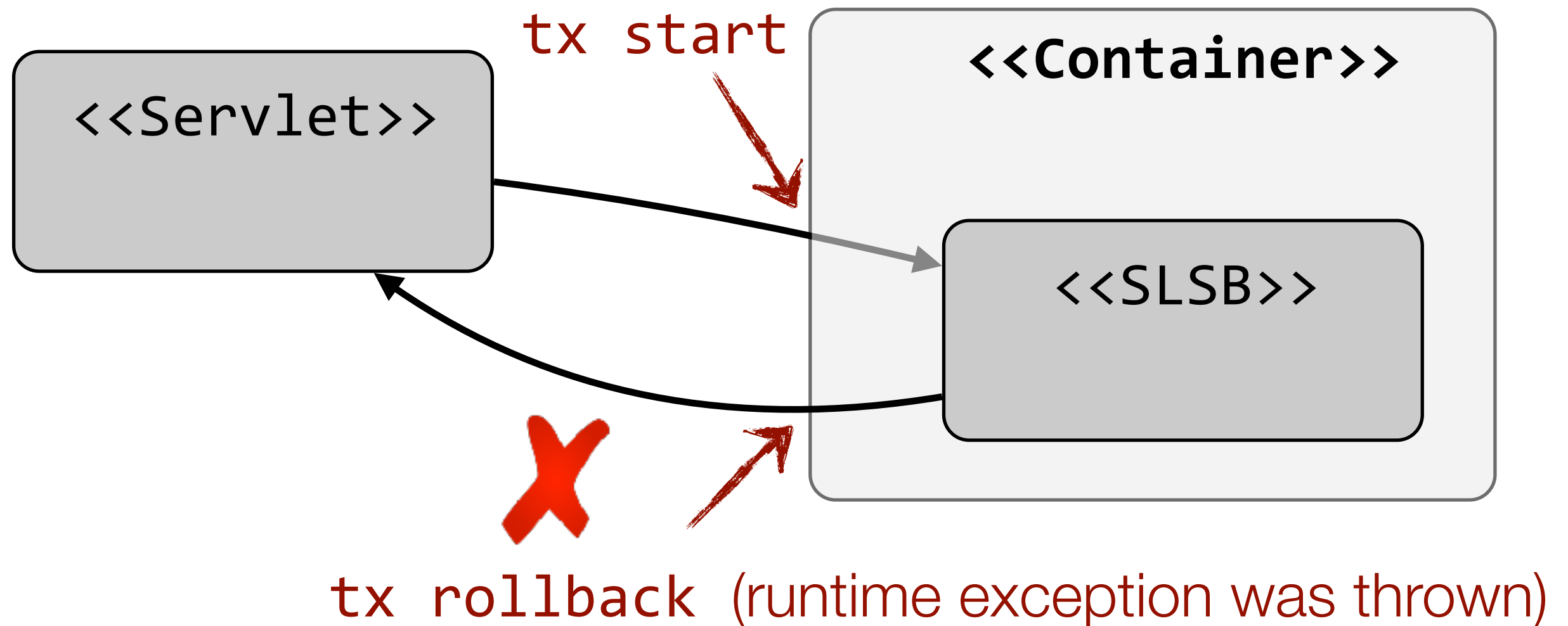
- By default, the EJB container handles calls to `commit` and `rollback`.
- Methods defined on EJBs provide demarcation points.
- This is the **default behavior**.



Transactions & EJBs



Transactions & EJBs



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 **throws an exception?**

Transaction Scope

What happens when a **client** calls a method on a session bean,

Opinion1

Everything should be rolled back!

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

Opinion2

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

which **throws an exception?**

Transaction Scope

What happens when a **client** calls a method on a session?

Opinion1

Everything should be rolled back!

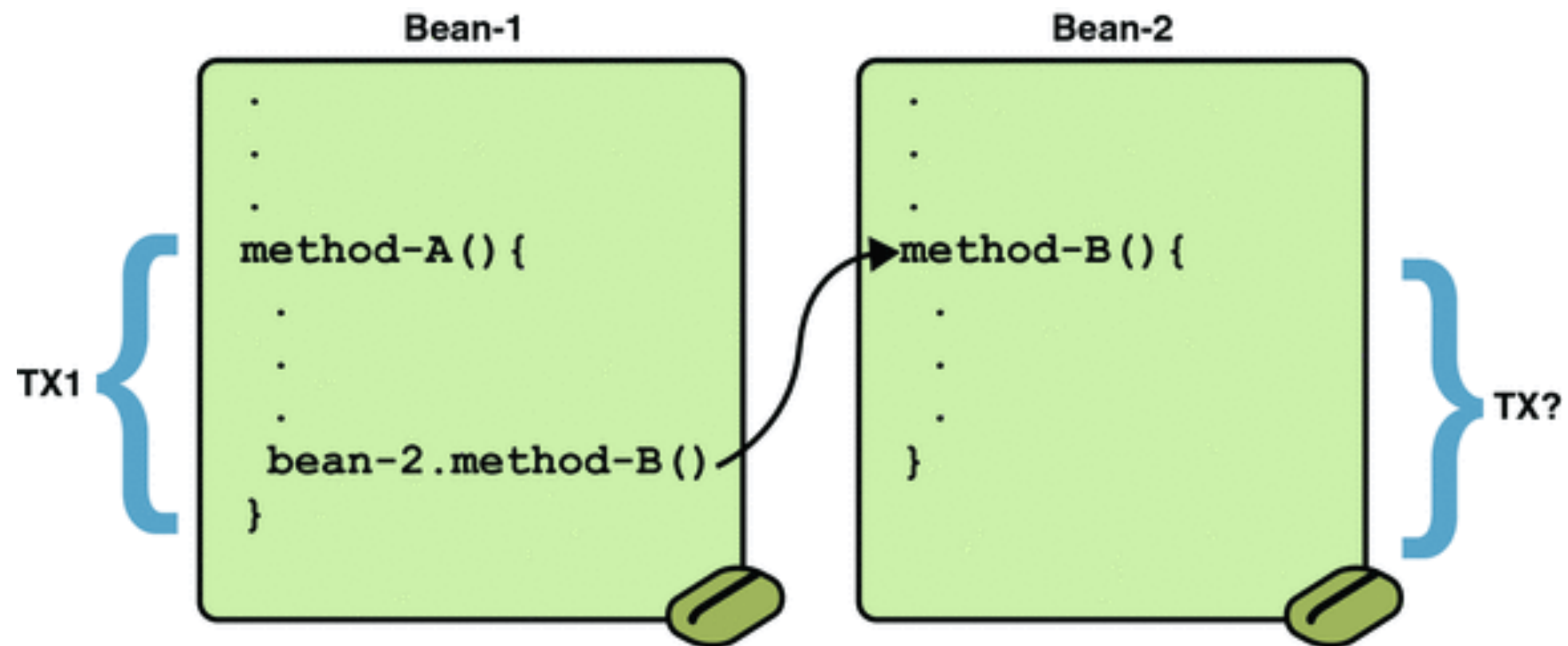
It is **up to the application** to specify intended behavior. The developer must specify transaction scope, typically with **annotations**.

Opinion2

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

which **throws an exception**?

Transaction Scope



<http://java.sun.com/javase/5/docs/tutorial/doc/bncij.html>

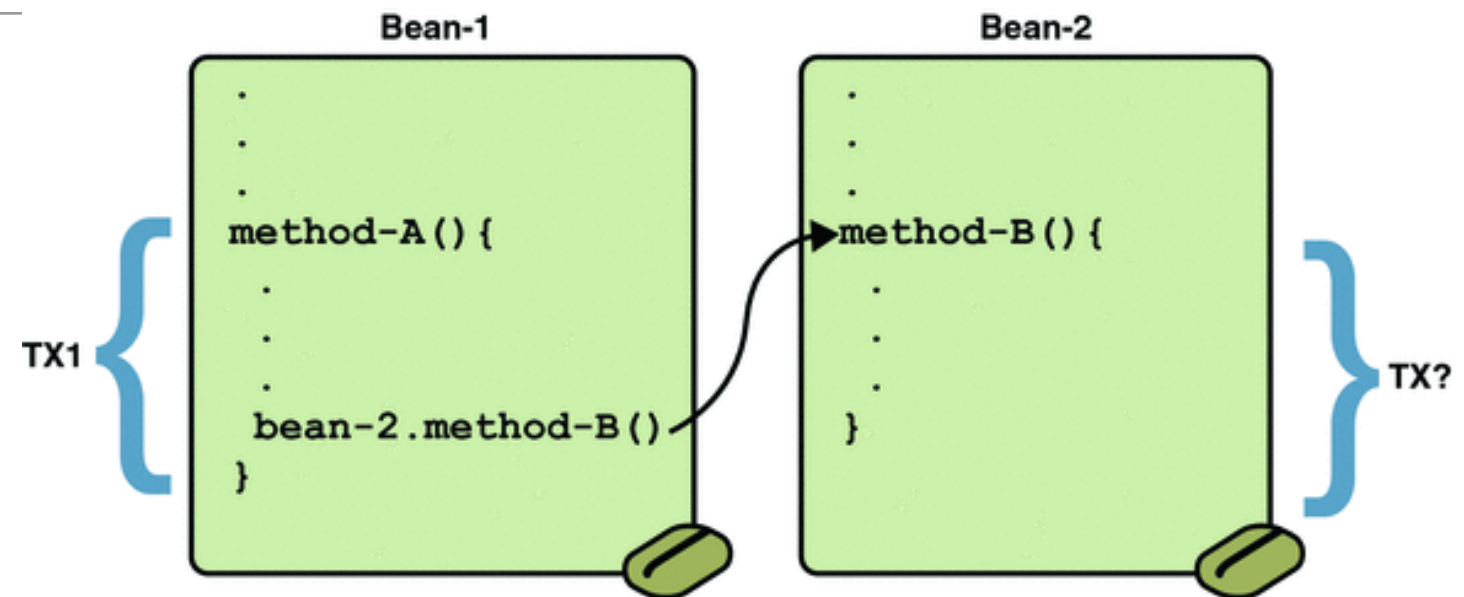
Transaction Scope

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

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

    public void thirdMethod() {...}

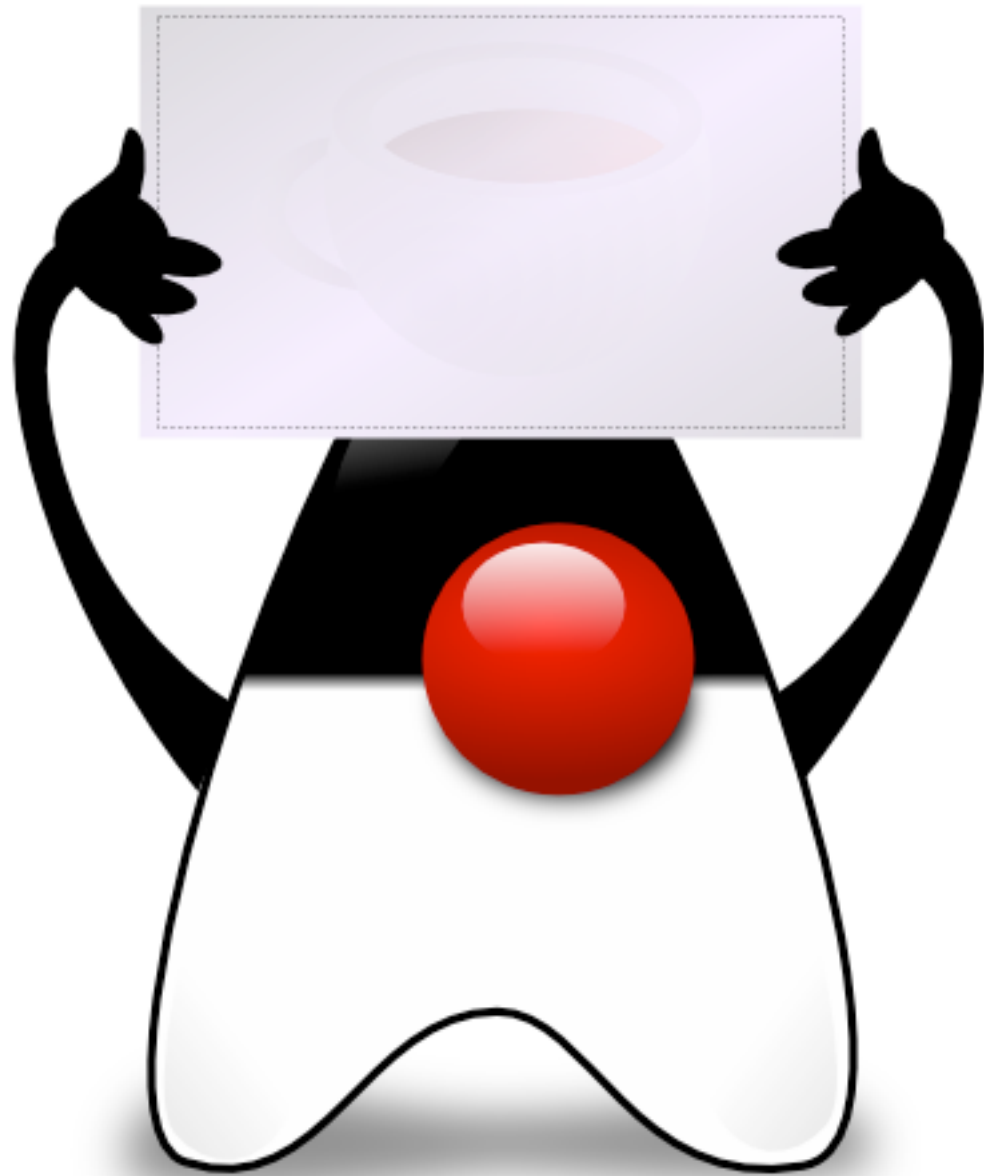
    public void fourthMethod() {...}
}
```



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**
- Firstly, if a **system exception** is thrown, the container will automatically roll back the transaction.
- Secondly, 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)**



JDBC without copy-paste



JDBC is pretty straightforward, but... isn't it **verbose and repetitive**?

```
@Stateless
public class SensorJdbcDAO implements SensorDAOLocal {

    @Resource(lookup = "jdbc/AMTDatabase")
    private DataSource dataSource;

    public List<Sensor> findAll() {
        List<Sensor> result = new LinkedList<>();
        try {
            Connection con = dataSource.getConnection();

            PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
            ResultSet rs = ps.executeQuery();

            while (rs.next()) {
                Sensor sensor = new Sensor();
                sensor.setId(rs.getLong("ID"));
                sensor.setDescription(rs.getString("DESCRIPTION"));
                sensor.setType(rs.getString("TYPE"));
                result.add(sensor);
            }

            ps.close();
            con.close();

        } catch (SQLException ex) {
            Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return result;
    }
}
```

When I implement the UserDAO, the RoleDAO, the LocationDAO, will I need to **repeat** all the code around those statements (**boilerplate**)?

Will I need to manually replace the **table and column names** in each DAO?

And when I **maintain** my application, what happens when a new property is added? Do I have to **update my DAO**?


```
@Stateless
public class SensorJdbcDAO implements SensorDAOLocal {

    @Resource(lookup = "jdbc/AMTDatabase")
    private DataSource dataSource;

    public List<Sensor> findAll() {
        List<Sensor> result = new LinkedList<>();
        try {
            Connection con = dataSource.getConnection();

            PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
            ResultSet rs = ps.executeQuery();

            while (rs.next()) {
                Sensor sensor = new Sensor();
                sensor.setId(rs.getLong("ID"));
                sensor.setDescription(rs.getString("DESCRIPTION"));
                sensor.setType(rs.getString("TYPE"));
                result.add(sensor);
            }

            ps.close();
            con.close();
        } catch (SQLException ex) {
            Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
        }
        return result;
    }
}
```

Reflection sounds cool. Can't we use it to deal with JDBC in **more generic** ways?

JDBC gives me **metadata** about the DB schema.

Reflection gives me ways to dynamically find and **invoke methods** on Java objects.

Can we combine these features to make this code better?



What is the Java **Reflection** API?

- Reflection is a mechanism, through which a program can **inspect and manipulate** its structure and behavior **at runtime**.
- In Java, this means that a program can **get information about classes, their fields, their methods, etc.**
- In Java, this also means that a program can **create instances of classes dynamically** (based on their names, as in the example of JDBC drivers), **invoke methods**, etc.

`java.lang.Class`

`java.lang.reflect.Method`

`java.lang.reflect.Field`



Can you give me an example of **reflective code**?

- We can **load class definitions** and **create instances**, without hard-coding class names into Java identifiers:

```
Class dynamicManagerClass = Class.forName("ch.heigvd.amt.reflection.services.SensorsManager");  
Object dynamicManager = dynamicManagerClass.newInstance();
```

- For a class, we can **get the list of methods** and **their signature**:

```
Method[] methods = dynamicManagerClass.getMethods();  
  
for (Method method : methods) {  
    LOG.log(Level.INFO, "Method name: " + method.getName());  
  
    Parameter[] parameters = method.getParameters();  
    for (Parameter p : parameters) {  
        LOG.log(Level.INFO, "p.getName()+ ":" + p.getType().getCanonicalName());  
    }  
}
```

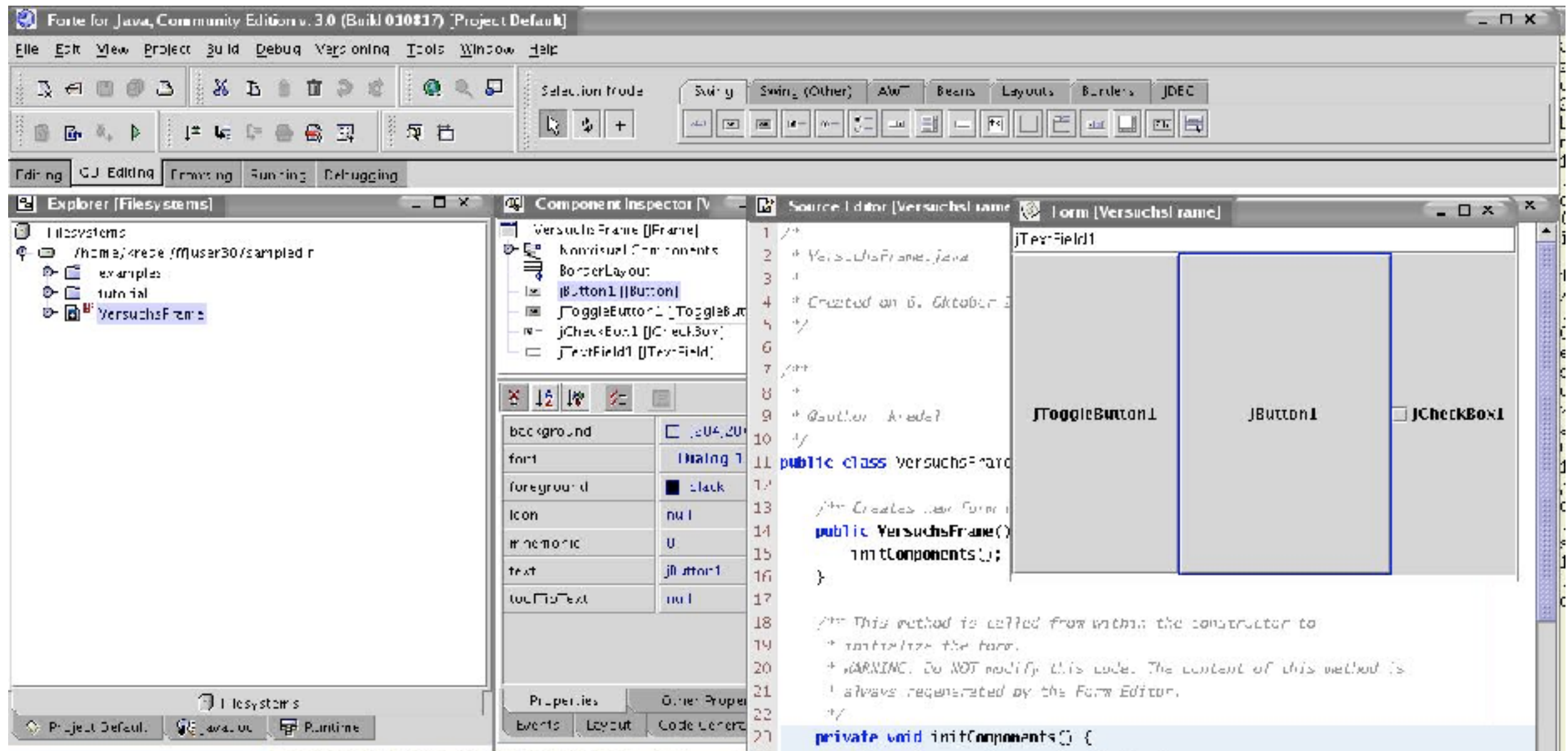
- We can dynamically **invoke a method** on an object:

```
Method method = dynamicManagerClass.getMethod("generateSensors", int.class, String.class);  
Object result = method.invoke(dynamicManager, 5, "hello");
```



What are **JavaBeans**?

- First of all, JavaBeans are **NOT** Enterprise Java Beans.
- The JavaBeans specification was proposed a very long time ago (1997) to enable the creation of **reusable components in Java**.
- One of the first use cases was to support the creation of **WYSIWYG development tools**. The programmer could drag and drop a GUI widget from a palette onto a window and edit its properties in a visual editor (think Visual Basic for Java).
- In this scenario, the GUI widgets would be packaged as JavaBeans by **third-party vendors**. The development tool would recognize them as such and would **dynamically extend the palette** of available components.



Forte for Java (aka Netbeans grand-father)



What are **JavaBeans**?

- Since then, JavaBeans have become **pervasive** in the Java Platform and are **used in many other scenarios**.
- This is particularly true in the Java EE Platform. Actually, **you have already implemented** JavaBeans without realizing it.
- While there are other aspects in the specification, the key elements are **coding conventions** that JavaBeans creators should respect:
 1. A JavaBean should have a **public no-args constructor**.
 2. A JavaBean should expose its properties via **getter** and **setter methods** with **well-defined names**.
 3. A JavaBean should be **serializable**.

```
public class Customer implements Serializable {
```

```
    public Customer() {}
```

```
    private String firstName;  
    private String lastName;  
    private boolean goodCustomer;
```

```
    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;  
    }
```

```
    public boolean isGoodCustomer() {  
        return goodCustomer;  
    }
```

```
    public void setGoodCustomer(boolean goodCustomer) {  
        this.goodCustomer = goodCustomer;  
    }
```

```
}
```

heig-vd

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

There is a **specific convention** for writing
getter methods for
boolean properties.



What are **JavaBeans**?

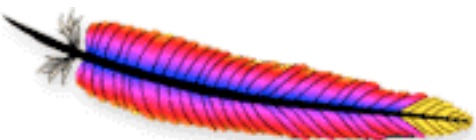
- These **coding and naming conventions** make it easier to **benefit from reflection** in **Java frameworks**:
 1. The framework can use the **public no-args constructor** to **create instances** with `Class.newInstance()`.
 2. The framework can **easily find out which methods it should call** (via reflection), based on a textual name. For instance, when a JSP page includes the string `${sensor.type}`, the runtime knows that it must invoke a method named "get" + "Type".
 3. The **state of a JavaBean** can travel over the wire (for instance when it moves from a remote EJB container to a web container).



What should I be know if I plan **to implement a framework** with JavaBeans?

- With the naming conventions defined in the JavaBeans specification, combined with Java reflection, **you can do pretty much everything yourself.**
- Have a look at the `java.beans` package and at the `Introspector` class. You will have easy access to properties, getters and setters.
- You should be aware of the **Apache Commons BeanUtils** library that will make your life easier.

*“The Java language provides **Reflection** and **Introspection** APIs (see the `java.lang.reflect` and `java.beans` packages in the JDK Javadocs). However, **these APIs can be quite complex** to understand and utilize. The BeanUtils component provides **easy-to-use wrappers** around these capabilities.”*



Apache CommonsTM
<http://commons.apache.org/>

<http://commons.apache.org/proper/commons-beanutils/>

commons
beanutilsTM



Back to the original question... How can I use reflection to **make my JDBC code generic**?

```
@Stateless  
public class SensorJdbcDAO implements SensorDAOLocal {
```

```
    @Resource(lookup = "jdbc/AMTDatabase")  
    private DataSource dataSource;
```

```
    public List<Sensor> findAll() {  
        List<Sensor> result = new LinkedList<>();  
        try {  
            Connection con = dataSource.getConnection();
```

```
            PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");  
            ResultSet rs = ps.executeQuery();
```

```
            while (rs.next()) {
```

```
                Sensor sensor = new Sensor();  
                sensor.setId(rs.getLong("ID"));  
                sensor.setDescription(rs.getString("DESCRIPTION"));  
                sensor.setType(rs.getString("TYPE"));  
                result.add(sensor);
```

```
            }
```

```
            ps.close();  
            con.close();
```

```
        } catch (SQLException ex) {  
            Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);  
        }  
        return result;  
    }
```

Reflection sounds cool. Can't we use it to deal with JDBC in **more generic** ways?

JDBC gives me **metadata** about the DB schema.

Reflection gives me ways to dynamically find and **invoke methods** on Java objects.

Can we combine these features to make this code better?



Back to the original question... How can I use reflection to **make my JDBC code generic**?

```
Sensor sensor = new Sensor();
sensor.setId(rs.getLong("ID"));
sensor.setDescription(rs.getString("DESCRIPTION"));
sensor.setType(rs.getString("TYPE"));
result.add(sensor);
```

Object-Relational Mapping in this example:

Table name = Class name + "s"

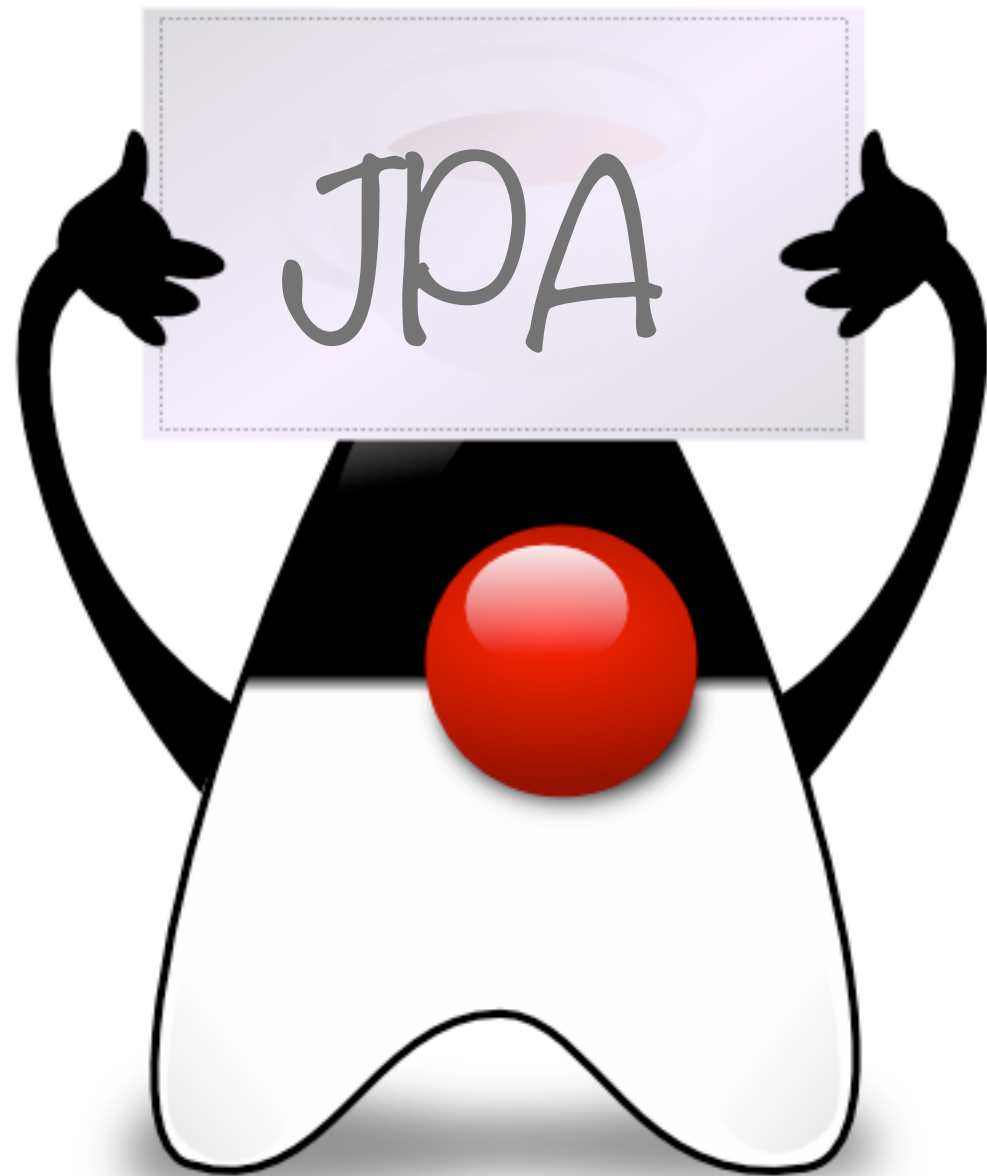
Column name = property name

```
String entityName = "Sensor";
String className = "ch.heigvd.amt.lab1.model." + entityName;
String tableName = entityName + "s";
PreparedStatement ps = con.prepareStatement("SELECT * FROM " + tableName);
ResultSet rs = ps.executeQuery();
Class entityClass = Class.forName(className);
PropertyDescriptor[] properties =
    Introspector.getBeanInfo(entityClass).getPropertyDescriptors();

while (rs.next()) {
    Object entity;
    entity = entityClass.newInstance();
    for (PropertyDescriptor property : properties) {
        Method method = property.getWriteMethod();
        String columnName = property.getName();
        try {
            method.invoke(entity, rs.getObject(columnName));
        } catch (SQLException e) {
            LOG.warning("Could not retrieve value for property " + property.getName()
                + " in result set. " + e.getMessage());
        }
    }
    result.add(entity);
}
```

Class names, property names, table names and column names do not have to be hard-coded.

What we need is a **mapping**. We can either rely on **conventions** or define it **explicitly**.



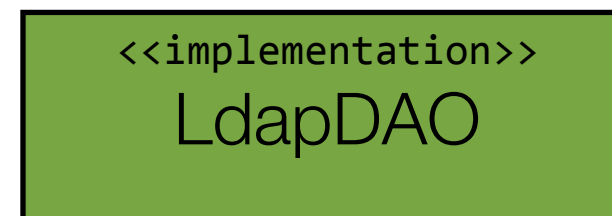
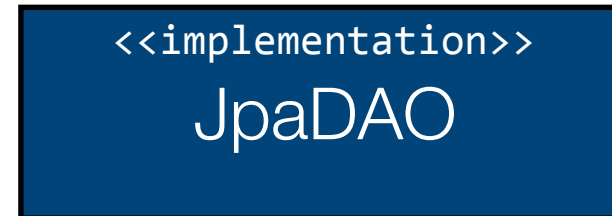
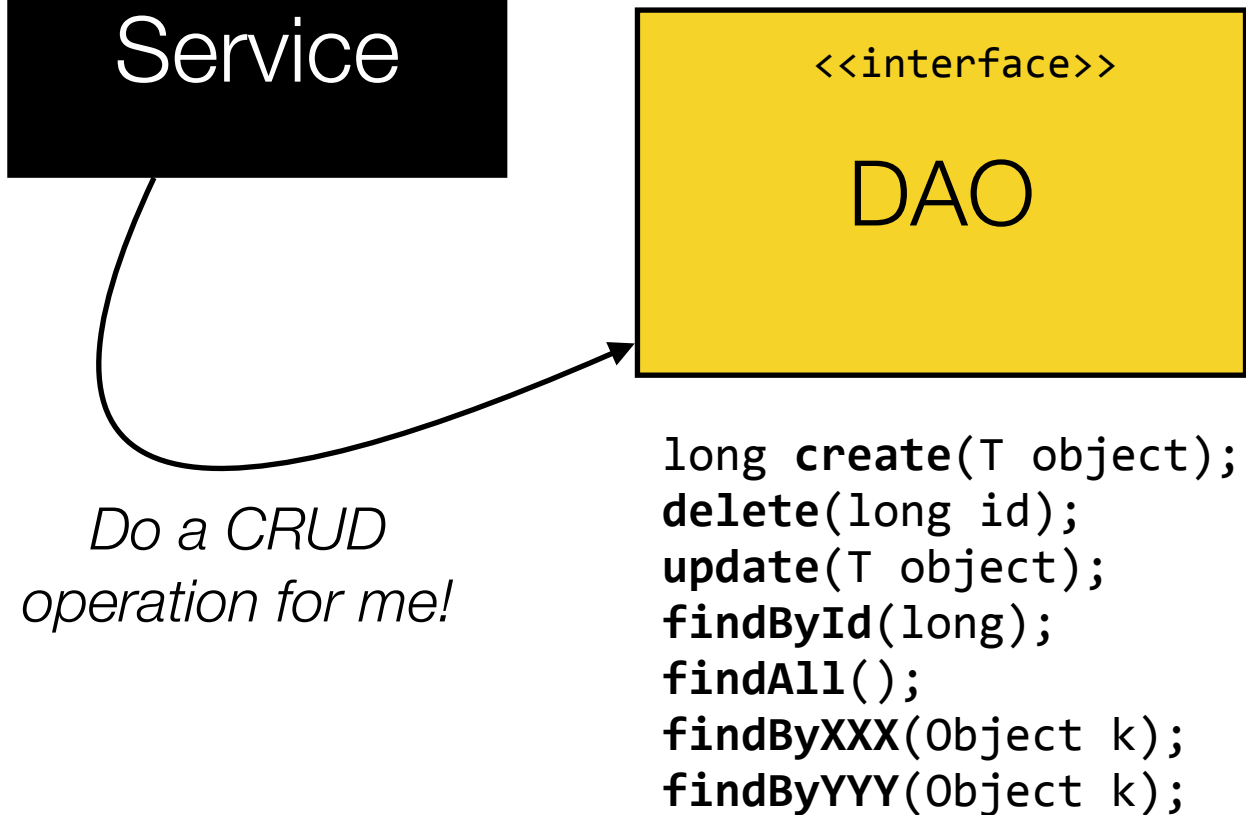
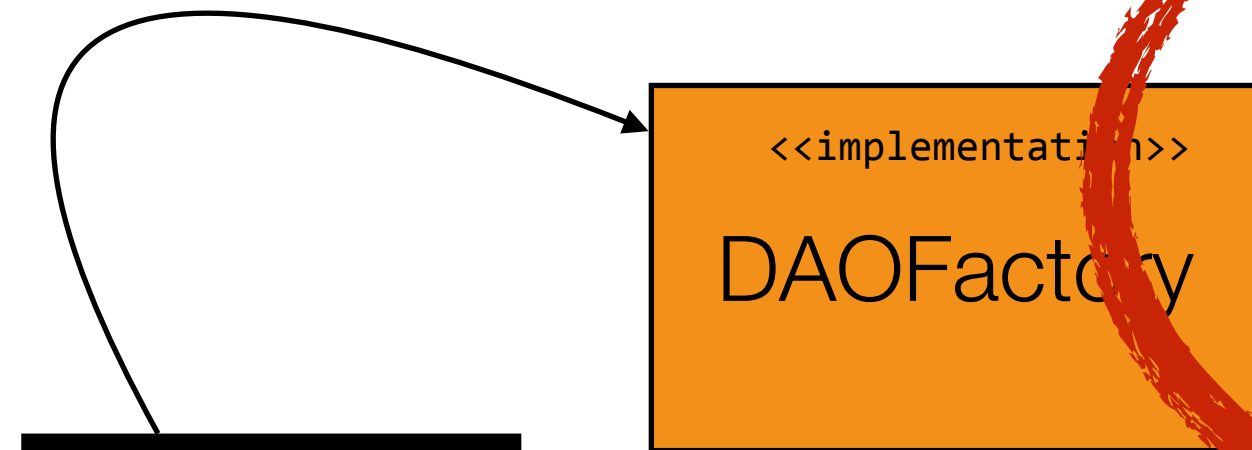
Java Persistence API (JPA)



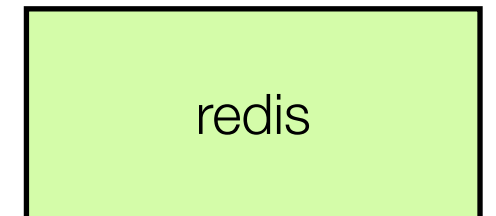
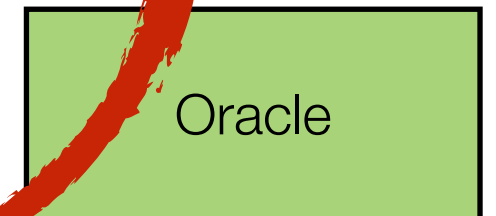
Foreword

- Today, we will **introduce** the **JPA API** and **related concepts**.
- In **future lectures**, we will come back and look at the details of transactions, associations, JPQL queries, etc.
- The objectives after today's lecture are:
 - to be able to **implement a simple JPA entity** (without associations)
 - to be able to **inject and use an EntityManager** in a **DAO**, in order to **create, update, delete** and **find** information in the database.
 - to write **simple JQPL queries** in the entity class, in order to write the finder methods.

Give me a DAO implementation!



heig-vd





What is **JPA**?

- JPA is a **standard API** for accessing **RDMS** from Java applications.
- JPA is a **higher-level API** than JDBC and is based on Object-Relational Mapping (ORM).
- When you use JPA, you do not have to write all SQL queries sent to the database. They are **automatically generated** by the JPA implementation.
- Java Persistence API was originally specified in JSR 220 (JPA 2.0 in JSR 317, JPA 2.1 in JSR 338).
- **JPA is part of Java EE**. This means that every compliant application server **has to** provide a JPA implementation.
- Historically, the design of JPA was greatly influenced by the **Hibernate framework**. Hibernate is one of the most popular JPA implementations (and provides features that are outside the scope of the specifications).

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.

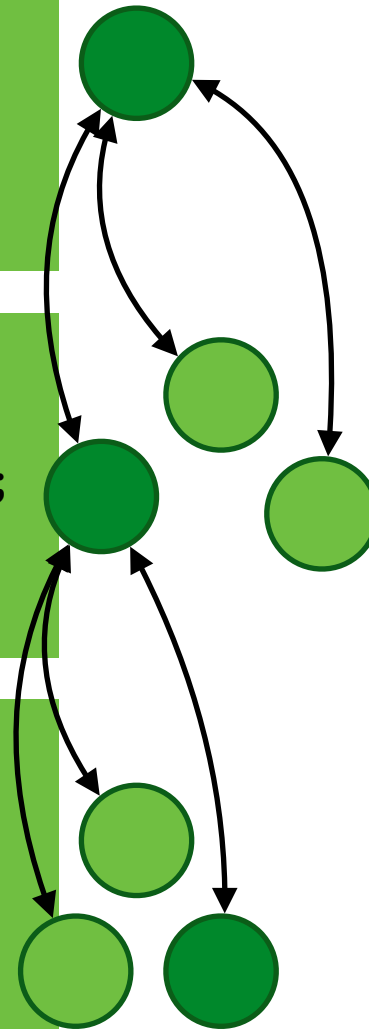


How do we move between the world of **objects** and the world of **relations**?

```
public interface ICustomer {  
    public String getFirstName();  
    public String getLastName();  
    public List<Order> getOrders();  
}
```

```
public interface IOrder {  
    public ICustomer getCustomer();  
    public List<OrderLine> getLines();  
    public double getTotal();  
}
```

```
public interface IOrderLine {  
    public Order getOrder();  
    public double getUnitPrice();  
    public double getQuantity();  
    public double getItemRef();  
}
```



CUSTOMER

ID	FIRSTNAME	LASTNAME	EMAIL
101	Olivier	Liechti	x.x@x.com
102	John	Doe	j.d@x.com
103	Paul	Smith	p.s@x.com

ORDER

ID	CUST_ID	DATE	TOTAL	STATUS
10230	101	02.03.2014	122.30	SHIPPED
20983	101	13.06.2014	256.00	SHIPPED
22099	101	18.07.2014	78.50	SHIPPED

ORDERLINE

ID	ORDER_ID	ITEM	QTY	U.PRICE
89123	20983	989	1	56
89124	20983	123	1	100
89125	20983	223	4	25

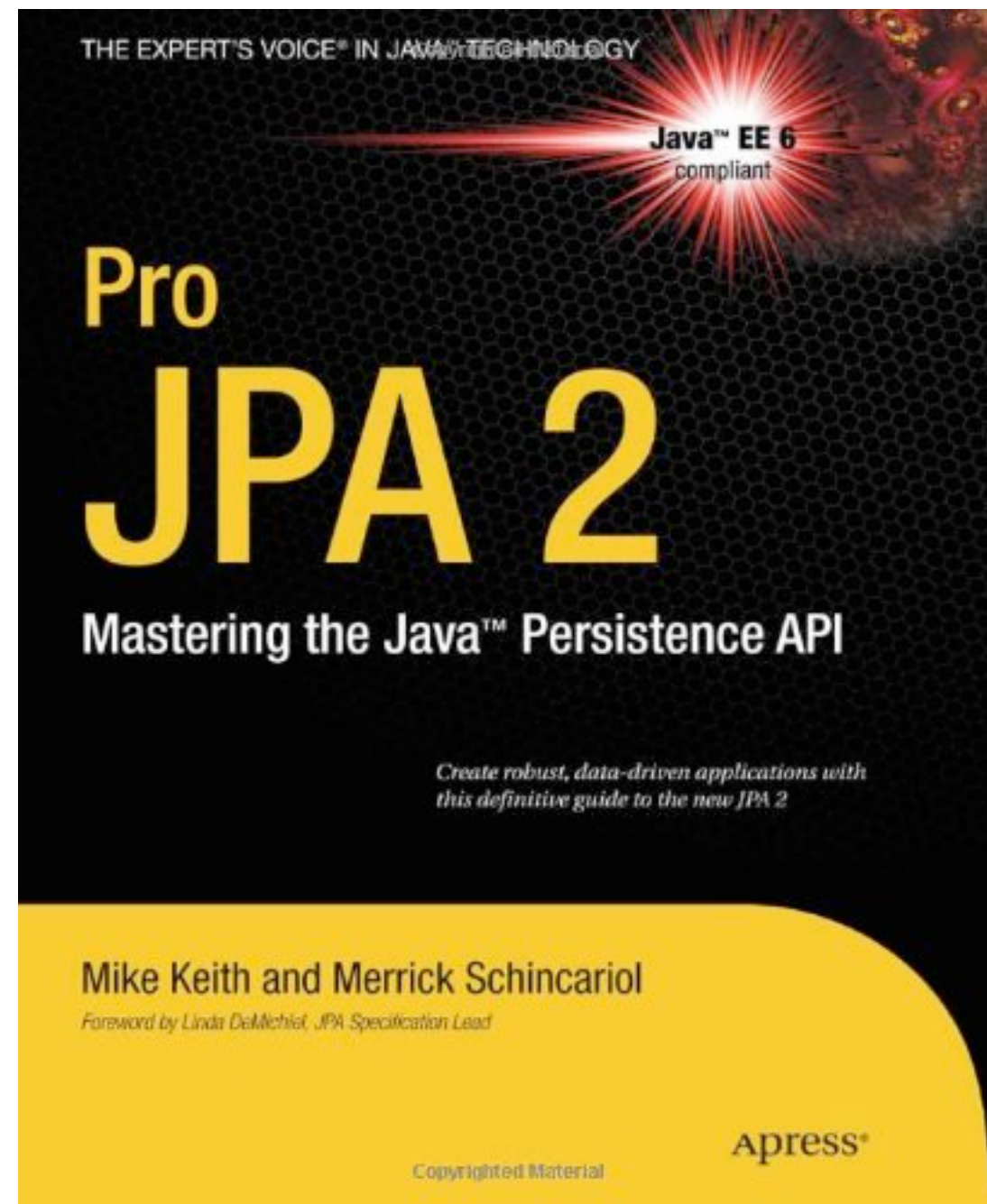
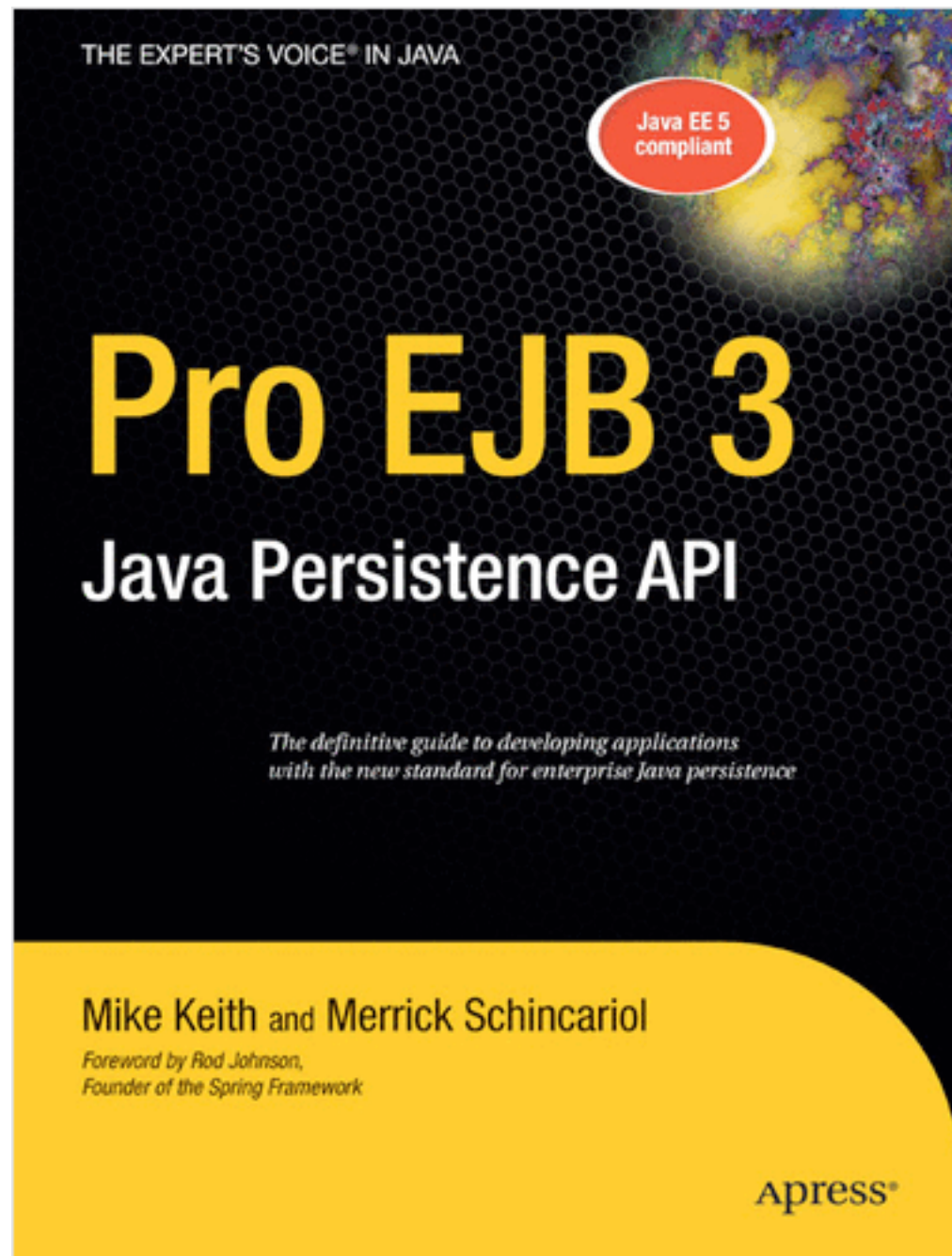
```
Customer customer = CustomersManager.findById(101);  
customer.getOrders().get(0).getLines().get(0).getItemRef();
```

```
SELECT ORDERLINE.ITEMREFERENCE  
FROM CUSTOMER  
INNER JOIN TABLE_ORDER on (TABLE_ORDER.CUSTOMER_ID = CUSTOMER.ID)  
INNER JOIN ORDERLINE on (TABLE_ORDER.ID = ORDERLINE.ORDER_ID)  
WHERE (CUSTOMER.ID = 101)  
LIMIT 1
```

Beware of large object graphs!

(they take time to load, they consume memory, they create contention on the DB, etc.)

The JPA Bible





How do I use **JPA** in my application?

- **Step 1 (static): 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 (dynamic): 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!



How do I use **JPA** in my application?

```
@Entity
public class Vehicle implements Serializable {
    @Id
    private long id;
    // properties, getters and setters
}
```

```
@Entity
public class Driver implements Serializable {
    @Id
    private long id;
    // properties, getters and setters
}
```

```
@Entity
public class Trip implements Serializable {
    @Id
    private long id;
    // properties, getters and setters
    @ManyToOne
    Driver driver;
    @ManyToOne
    Vehicle vehicle;
}
```

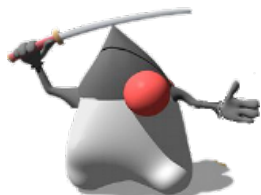
```
@Stateless
public class TripsManager {
    @PersistenceContext
    EntityManager em;

    public long createTrip(Trip trip) {
        em.persist(trip);
        em.flush();
        return trip.getId();
    }
}
```

INSERT INTO Trip (...) VALUES(...);

*With JPA, like with other Java EE API, you can rely on **conventions**. You don't have to explicitly specify all aspects. 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.*

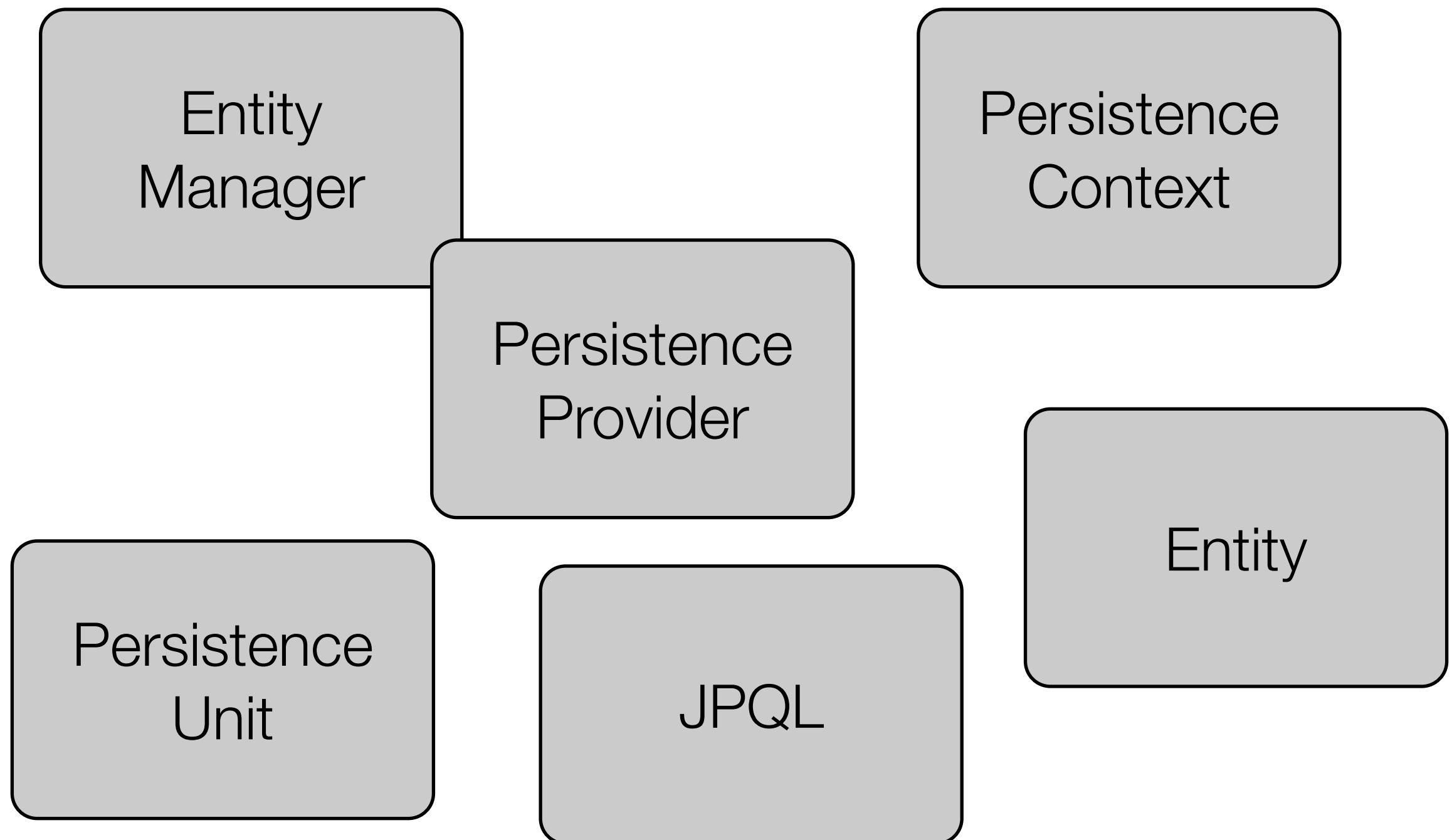


One frequent customization need arises when you have a business **entity name that collides with a SQL reserved word**. For instance, if you have a Role entity, MySQL will not allow you to create a table named Role. In that case, you will need to use the @Table(name="XXX") annotation.

*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.*

Abstractions defined in the JPA API



Persistence Provider

- A Persistence Provider is an **implementation** of the JPA API.
- **EclipseLink** and **Hibernate** are two examples of JPA Persistence Providers. EclipseLink is the one shipped with Glassfish.
- Persistence Providers are “**pluggable**”. This means that if you use only standard JPA features, you can for example decide to switch from EclipseLink to Hibernate at some point (Remember **SPI**?)
- 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. You have to **balance** functionality with portability...

- **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.

`@Entity` ← This is an entity class

```
public class Student implements Serializable {  
    private static final long serialVersionUID = 1L;
```

`@Id` ← An entity needs a unique id

`@GeneratedValue(strategy = GenerationType.AUTO)` ← There are different ways to generate these id values

```
    private Long id;  
  
    private String firstName;  
    private String lastName;
```

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

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

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

```
    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;  
    }
```

```
    ...  
}
```

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 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)  
...
```

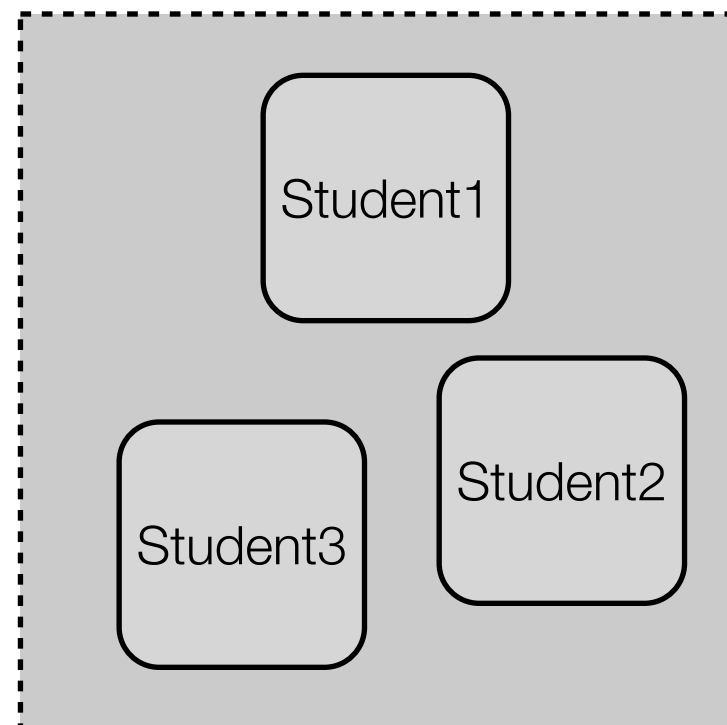
Using the Entity Manager

- 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.



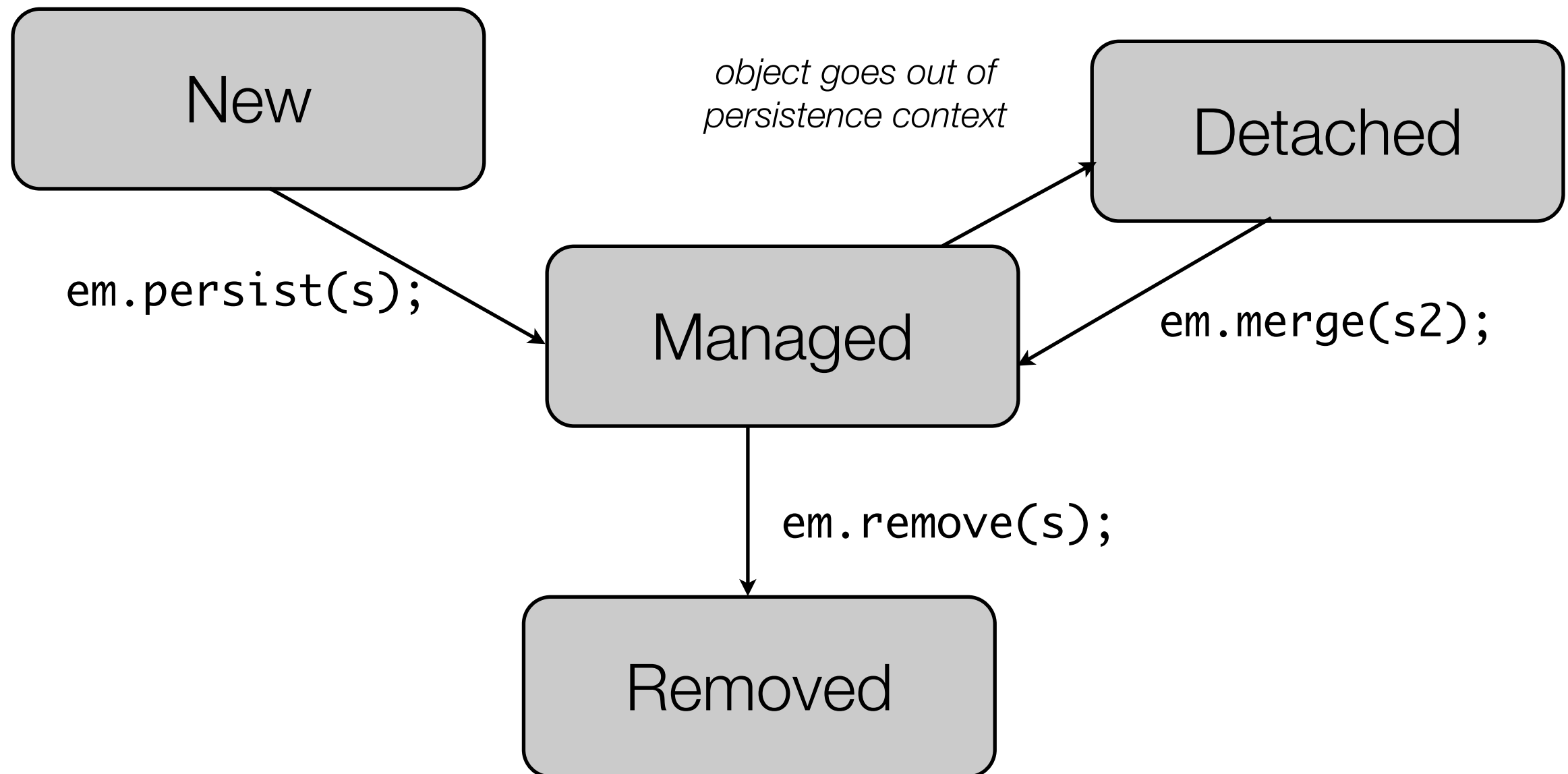
Persistence Context

- “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

```
Student s = new Student();
```

*Think what happens when
an EJB returns an object
to a servlet!*





When do objects enter and leave the persistence context?



The persistence context is **created** when **transaction** begins and is **flushed** when transaction commits (or rollbacks).



A **transaction** is started by the **EJB container** whenever a business method is called. It is committed by the container when it returns (or rolled back if there is an exception).

```
@Stateless
public class Manager {

    @PersistenceContext
    EntityManager em;

    public void businessMethod() {
        Customer c1 = new Customer();

        Customer c2 = new Customer();
        em.persist(c2);

        Customer c3 = em.find(123);

        Customer c4 = new Customer(246, "john", "doe");
        Customer c5 = em.merge(c4);
    }
}
```

breakpoint

JVM memory space

JPA Persistence context (managed entities)

c2

c3

c5

c1

c4





```
Customer c1 = new Customer();
```



Creating a new instance of a JPA entity does not make it a managed object. At this stage, it is a **simple POJO** that is not linked to the DB (*)

```
Customer c2 = new Customer();  
// c2 is not in persistence ctx
```



Calling **em.persist(c2)** brings c2 into the **persistence context**. From this point, JPA intercepts all calls made to c2. So, it knows when c2 is modified by a client (i.e. when it becomes “**dirty**”).

```
em.persist(c2);  
// c2 is in the persistence ctx
```



Note that at this stage, it is most likely that **nothing has been written to the DB**. SQL statements will only be issued when the transaction commits.

```
Customer c3 = em.find(123);
```



Calling **em.find(123)** issues a SELECT query to the DB. An entity is created with the result and is **brought into the persistence context**.

```
Customer c4 = new  
    Customer(246, “john”, “doe”);  
Customer c5 = em.merge(c4);
```



c4 is a simple POJO. When **em.merge(c4)** is invoked, a SELECT statement will be issued to retrieve a row where the primary key is equal to 246. A new entity is created and its properties are copied from c4 (to update the DB later on). **WARNING: c5 is in the persistence context, c4 is not!!**

(*) Sort of... see a description of how the magic can happen: <http://struberg.wordpress.com/2012/01/08/jpa-enhancement-done-right/>

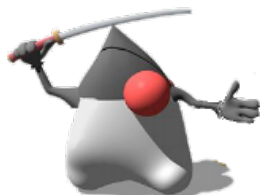
Persistence Context Types

- 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”.

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

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



The developer has the responsibility to “wire” both sides of bi-directional relationships. You will forget to do that. You will not get an immediate error. You will see weird behavior and spend at least 2 hours debugging this.

Beware of bi-directional associations!

They make it easy to create bugs (if developers forget to connect both sides). They are rarely necessary, if useful at all.

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;
}
```



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)



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- 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)



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- 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

Entity
Manager

I am an interface that components use to interact with the persistence service (CRUD, queries)

Persistence
Context

I am a set of entity instances, at runtime. Often, I live as long as a transaction.

Persistence
Provider

I do the work. I am the implementation of the JPA API. I am Toplink, Hibernate, etc.

Entity

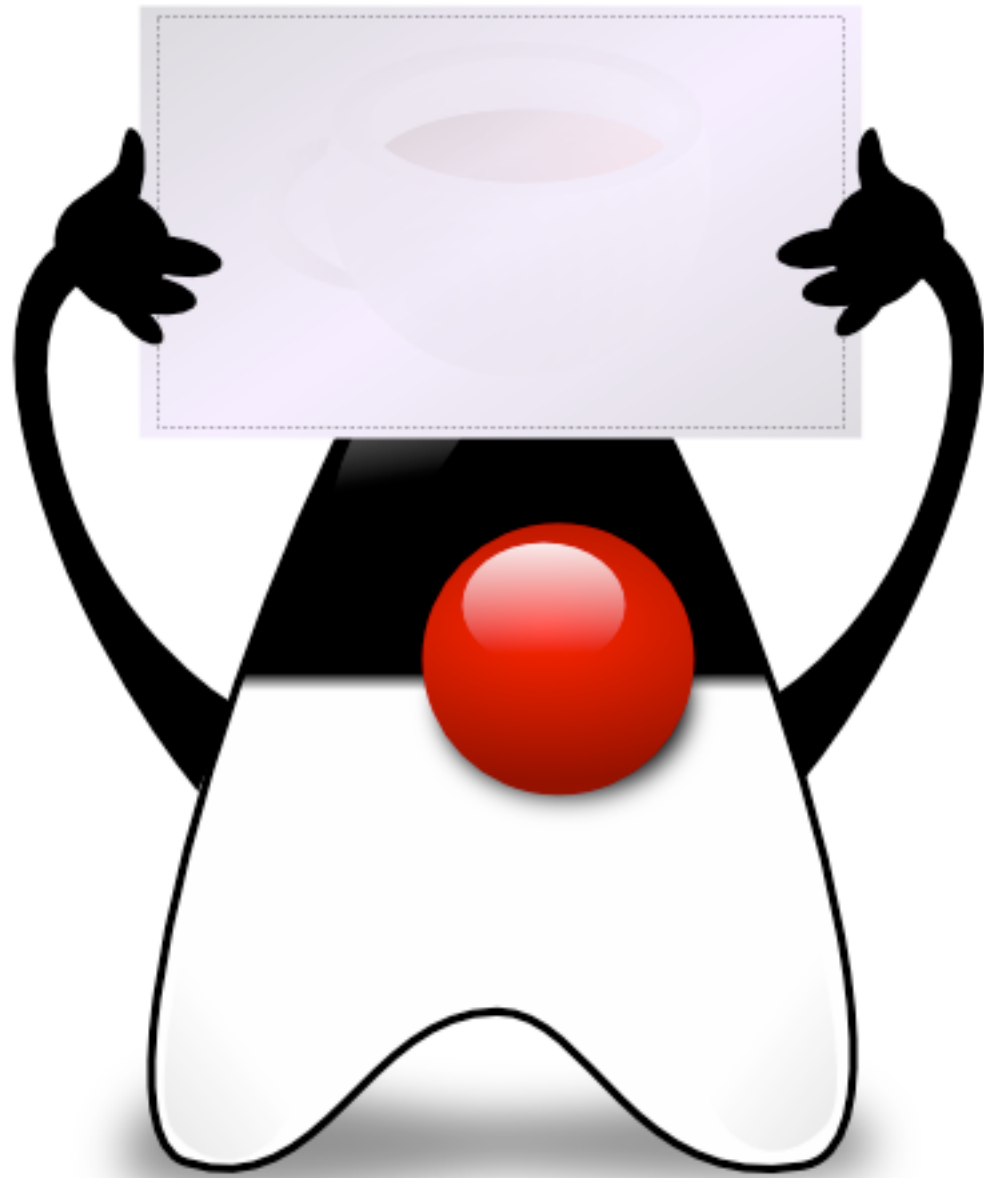
I am a business object, my state will be transparently stored in a database. I am a POJO.

Persistence
Unit

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.

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 (with EJBs/JPA)

Transaction scope & JPA

What happens if there is a strike and a **NullPointerException** is thrown in the constructor?

```
@Stateless
public class CarService {

    @PersistenceContext
    EntityManager em;

    @EJB
    PartsService partsService;

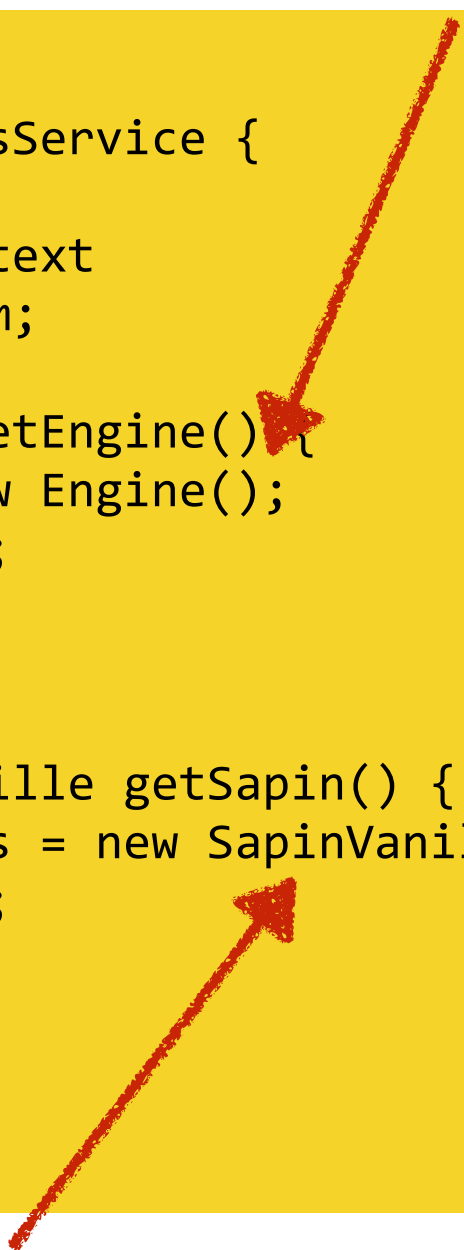
    public Car buildCar() {
        Engine e = getEngine();
        SapinVanille s = getSapin();
        Car c = new Car(e, s);
        em.persist(c);
    }
}
```

```
@Stateless
public class PartsService {

    @PersistenceContext
    EntityManager em;

    public Engine getEngine() {
        Engine e = new Engine();
        em.persist(e);
        return e;
    }

    public SapinVanille getSapin() {
        SapinVanille s = new SapinVanille();
        em.persist(s);
        return(s);
    }
}
```

Two red arrows are drawn on the code block. One arrow points from the top right towards the getEngine() method, and the other points from the bottom right towards the getSapin() method.

What happens if there is a shortage of vanilla and a **NullPointerException** is thrown in the constructor?

Transaction scope & JPA

- The default transaction scope for EJB methods is “REQUIRED”. This means that we will have one single transaction for the whole process (and one JPA persistence context).
- **Scenario 1:** no exception thrown. The 3 rows will be inserted when the container commits.

T1	Persistence Context
<code>[CarService] Engine e = getEngine();</code>	<code>PC[t1]= {}</code>
<code>[PartsService] Engine e = new Engine(); em.persist(e);</code>	<code>PC[t1]= {e}</code>
<code>[CarService] SapinVanille s = getSapin();</code>	<code>PC[t1]= {e}</code>
<code>[PartsService] SapinVanille s = new SapinVanille(); em.persist(s);</code>	<code>PC[t1]= {e, s}</code>
<code>[CarService] Car c = new Car(e, s); em.persist(c);</code>	<code>PC[t1]= {e, s, c}</code>

Transaction scope & JPA

- The default transaction scope for EJB methods is “REQUIRED”. This means that we will have one single transaction for the whole process (and one JPA persistence context).
- **Scenario 2:** an exception is thrown in the SapinVanille constructor. No row is added to the database (not even the engine which was successfully persisted).

T1	Persistence Context
[CarService] Engine e = getEngine();	PC[t1]= {}
[PartsService] Engine e = new Engine(); em.persist(e);	PC[t1]= {e}
[CarService] SapinVanille s = getSapin();	PC[t1]= {e}
[PartsService] SapinVanille s = new SapinVanille(); NullPointerException is thrown	PC[t1]= {e}
Transaction is rolled back by the container	

Transaction scope & JPA

- Vanilla shortage should not block the production line!
- The Car constructor is ok with a null value anyway. Let's execute the getSapin() method in its own transaction.

```
@Stateless
public class CarService {

    @PersistenceContext
    EntityManager em;

    @EJB
    PartsService partsService;

    public Car buildCar() {
        Engine e = getEngine();
        try {
            SapinVanille s = getSapin();
        } catch (Exception e) {
            logException(e);
        }
        Car c = new Car(e, s);
        em.persist(c);
    }
}
```

```
@Stateless
public class PartsService {

    @PersistenceContext
    EntityManager em;

    public Engine getEngine() {
        Engine e = new Engine();
        em.persist(e);
        return e;
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public SapinVanille getSapin() {
        SapinVanille s = new SapinVanille();
        em.persist(s);
        return(s);
    }
}
```

Transaction scope & JPA

- **Scenario 1:** no exception thrown. 1 row is committed in the SapinVanille table first, 2 rows are committed in the Engine and Car tables later.

s is not a managed entity

T1	T2	Persistence Context
[CarService] Engine e = getEngine();		PC[t1]= {}
[PartsService] Engine e = new Engine(); em.persist(e);		PC[t1]= {e}
[CarService] SapinVanille s = getSapin();		PC[t1]= {e}
	[PartsService] SapinVanille s = new SapinVanille(); em.persist(s);	PC[t2]= {s}
	The container commits T2	
[CarService] Car c = new Car(e, s); em.persist(c);		PC[t1]= {e, c}
	The container commits T1	

Transaction scope & JPA

- **Scenario 2:** an exception is thrown in the SapinVanille constructor. No row is committed in the SapinVanille table, BUT 2 rows are committed in the Engine and Car tables!

T1	T2	Persistence Context
[CarService] Engine e = getEngine();		PC[t1]= {}
[PartsService] Engine e = new Engine(); em.persist(e);		PC[t1]= {e}
[CarService] SapinVanille s = getSapin();		PC[t1]= {e}
	[PartsService] SapinVanille s = new SapinVanille(); NullPointerException is thrown	PC[t2]= {}
	The container rolls back T2	
[CarService] Car c = new Car(e, s); em.persist(c);		PC[t1]= {e, c}
	The container commits T1	


Transactions & concurrency control

- If several transactions are processed **concurrently**, unexpected results may occur. There are different strategies and mechanisms for dealing with that.

```
@Stateless
public class TransactionProcessor {

    @EJB
    AccountDAO accountDao;

    public void processTransaction(Transaction t) {
        Account a = accountDao.findById(t.getAccountId());
        long previousBalance = a.getBalance();
        a.setBalance(previousBalance + t.getAmount());
    }
}
```

A red arrow points from the right side of the slide to the '+' sign in the line 'a.setBalance(previousBalance + t.getAmount());', highlighting the point of potential concurrency issues.

What happens if another transaction modifies the account balance between these two statements?

Optimistic concurrency control



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- In many applications, there is a **high ratio of “read to write” operations** (many transactions read data, few update data). Moreover, there is a “small” likelihood that two concurrent transactions try to update the same data.
- In this case, for performance and scalability reasons, it is often recommended to implement an optimistic concurrency control mechanism.
- The mechanism works as follows:
 - When a program **reads** a record, it gets its “**version number**” (the number of previous updates) in a table column.
 - When it **updates** this record, it makes sure that the version number has not been incremented (this would indicate a conflict with another transaction).
- The developer has to write the logic to execute when a conflict is notified (retry, notify the user, etc.)

Optimistic concurrency control with JPA



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- **JPA supports optimistic concurrency control.**
- To use it, the first step is to annotate one field of the entity with the **@version** annotation. JPA will ensure that this value is incremented with every update.
- The second step is to catch the **OptimisticLockException** that may be thrown by JPA when the transaction commits.
- This is where the developer specifies what to do if a conflict has been detected. In some cases, it is possible to immediately and silently retry the transaction.

```
@Entity
public class Account {

    @Id
    long accountId;

    @Version
    long version;
}
```

Pessimistic concurrency control

- When an optimistic concurrency control is not appropriate, then it is possible to implement **pessimistic concurrency control with locks**.
- RDBMS support different types of locks (read lock, write lock).
- When a transaction obtains a **read lock** on a record, it cannot be modified by other transactions. However, it can be read by other transactions.
- When a transaction obtains a **write lock** on a record, it cannot be modified, nor read by other transactions.
- Locking database records **introduce issues**: scalability, performance, deadlocks. It can be tricky to decide when to obtain a lock and for how long.

Pessimistic concurrency control with JPA



HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

- **JPA supports pessimistic concurrency control since version 2.0**
- It is possible to **lock a record** with **em.lock(entity, LOCK_TYPE)**.
- It is also possible to lock a record at the time of retrieval with **em.find(class, id, LOCK_TYPE)**

```
Account a = em.find(Account.class, id);  
em.lock(a, PESSIMISTIC_WRITE);
```

Be aware that we
still have a risk of
stale data here!

```
Account a = em.find(Account.class, id, PESSIMISTIC_WRITE);
```