Lecture 3: The Data Access Tier

Olivier Liechti
AMT

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

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

Today's agenda

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

14h00 - 15h00	60'	**Lecture** The Data Access Object (DAO) Pattern JDBC **Demo / Exercise** Setting up MySQL & Glassfish, running the updated MVCDemo project
15h00 - 15h10	*10'*	*Break*
15h10 - 16h25	75'	**Lecture** Java Persistence API (JPA) Java Reflection & JavaBeans **Demo / Exercise** Writing User Acceptance Tests with Selenium and the WebDriver API

heig-vd

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

The DAO Design Pattern

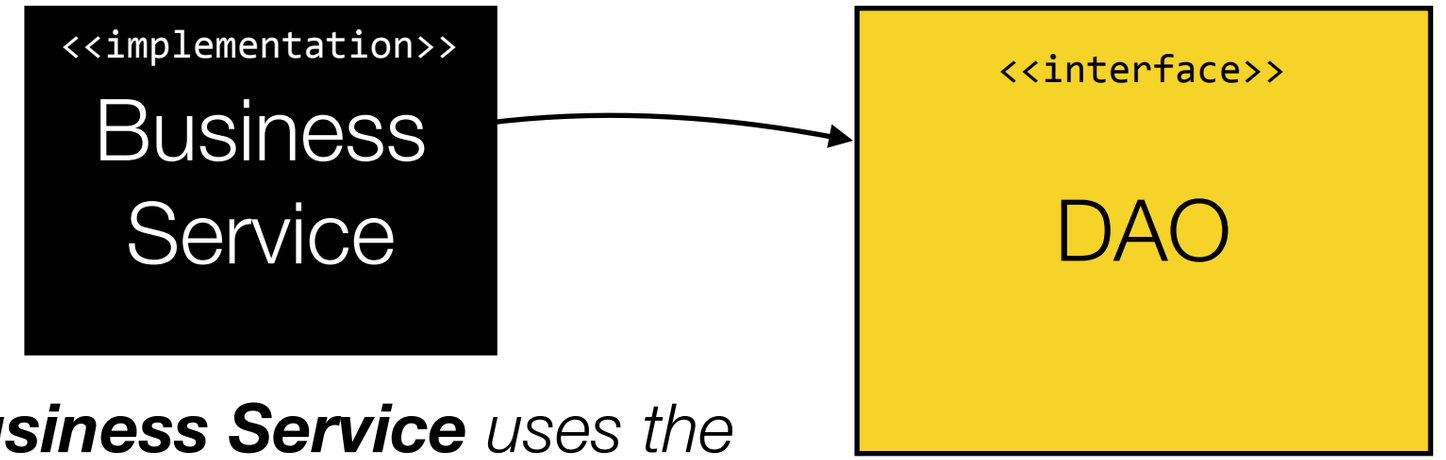
What is the **DAO design pattern** and what are its benefits?

- Most applications manipulate data that is stored in one or more **data stores**.
- There are **different ways** to implement a data store. Think about specific RDMS, NoSQL DBs, LDAP servers, file systems, etc.
- When you implement business logic, you would like to create code that is **independent** from a particular data store implementation (*).
- In other words, you want to **reduce coupling** between your business service and your data store implementation.
- When you apply the **Data Access Object** design pattern, you create an abstraction layer to achieve this goal.

(*) This is true only to some extent… you cannot completely forget about it, for instance for performance reasons

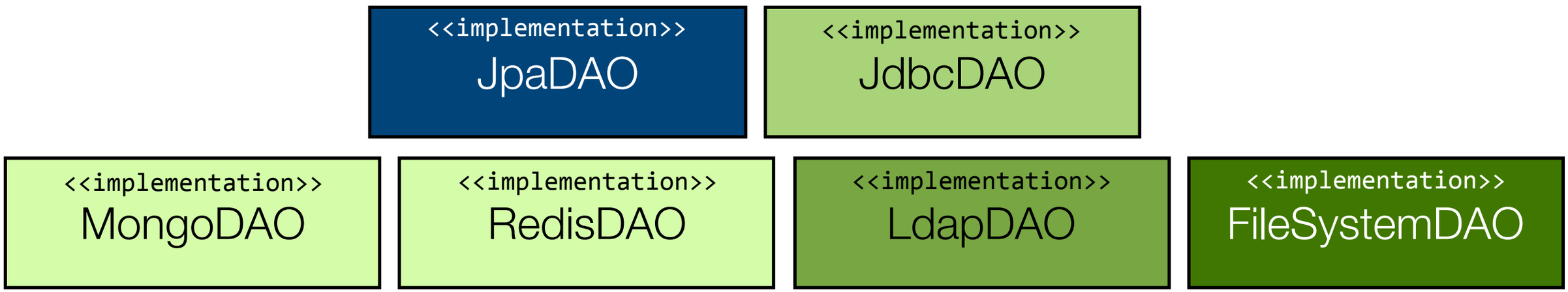
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

*The **DAO interface** defines generic **CRUD** operations and **finder** methods*

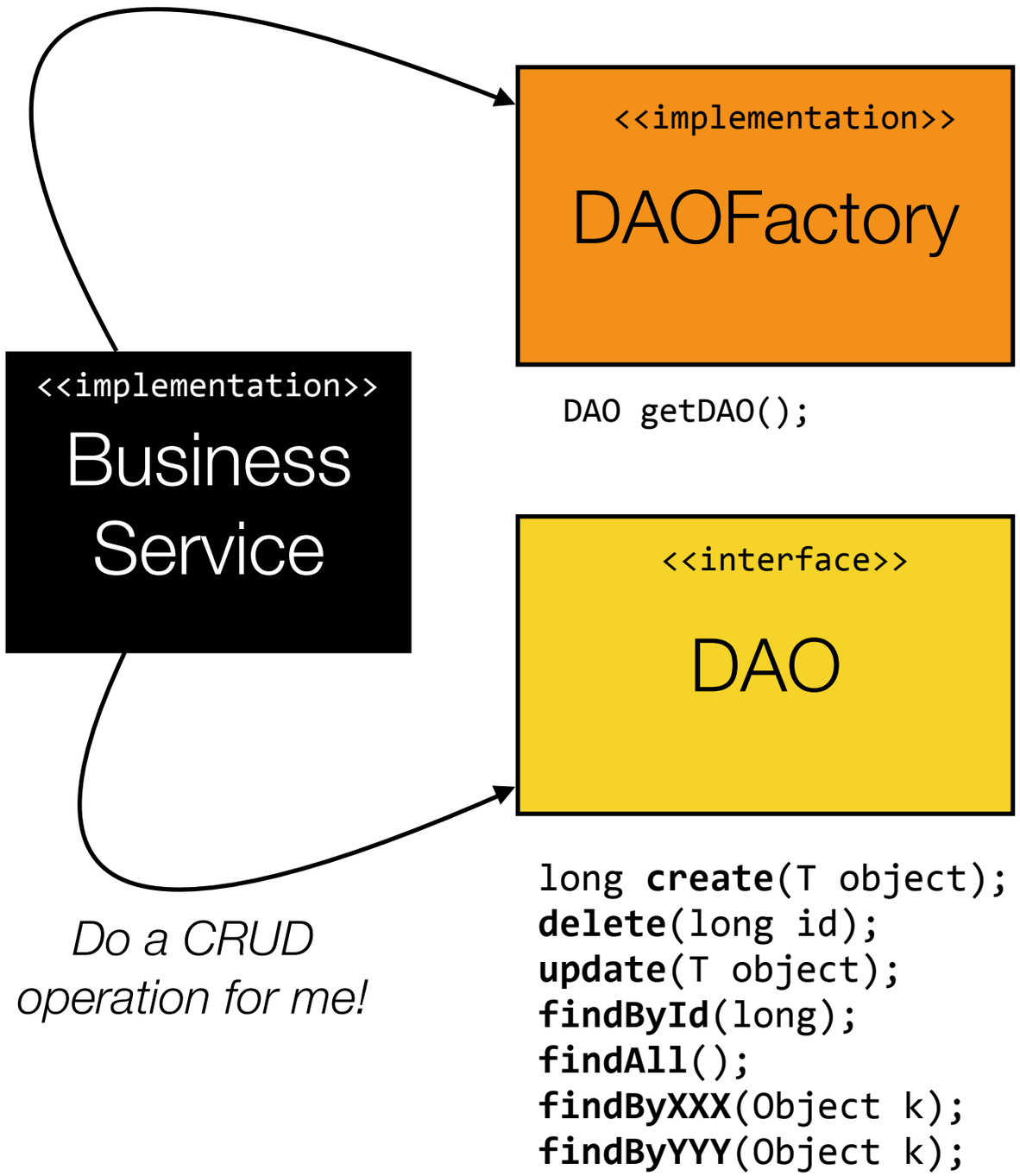


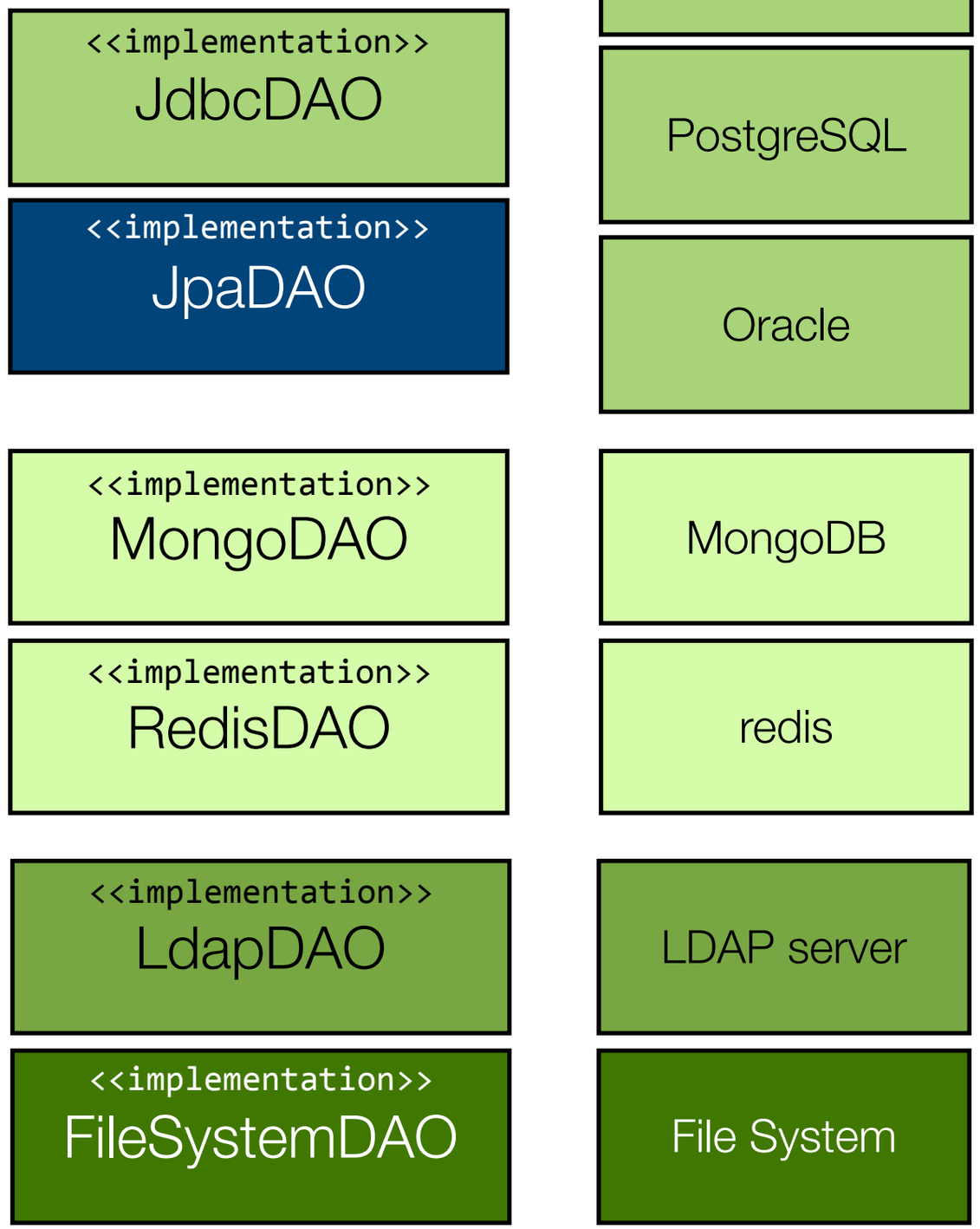
*The **Business Service** uses the DAO interface to interact with a particular DAO implementation*

***DAO implementations** handle interactions with specific data stores*



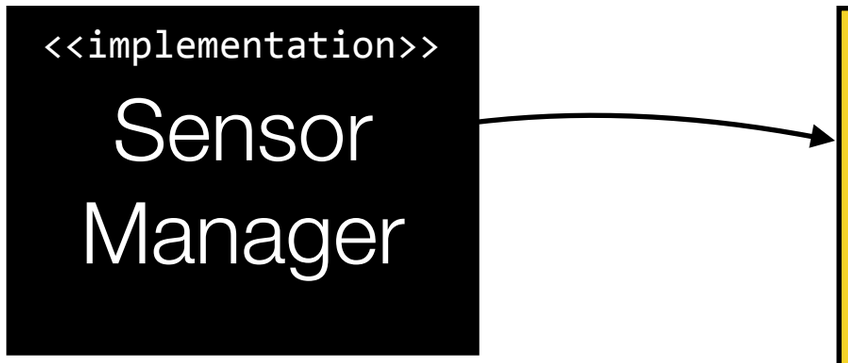
Give me a DAO implementation!

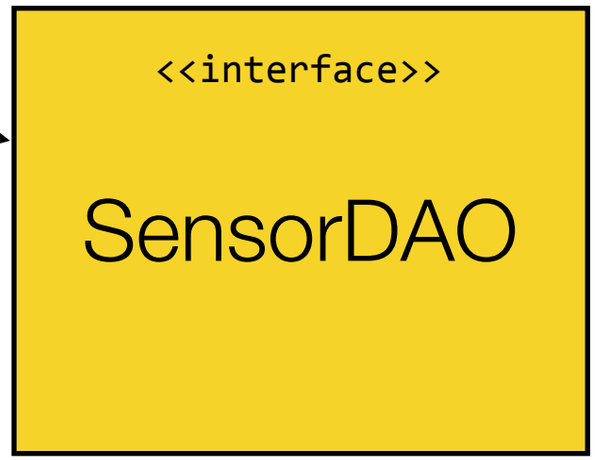




heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

*The **DAO interface** defines generic **CRUD** operations and **finder** methods*

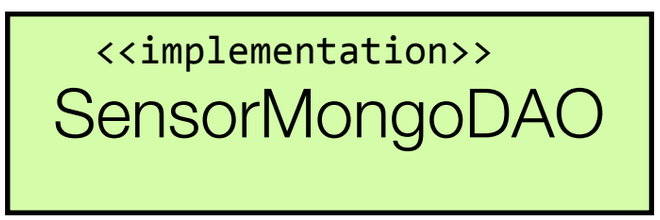


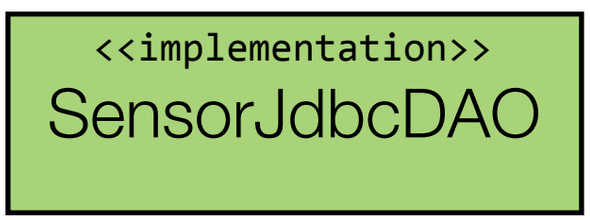



```
long create(Sensor s);
delete(long id);
update(T object);
findById(long);
findAll();
findByLocation(Location l);
findByType(SensorType t);
```

*The **Business Service** uses the DAO interface to interact with a particular DAO implementation*

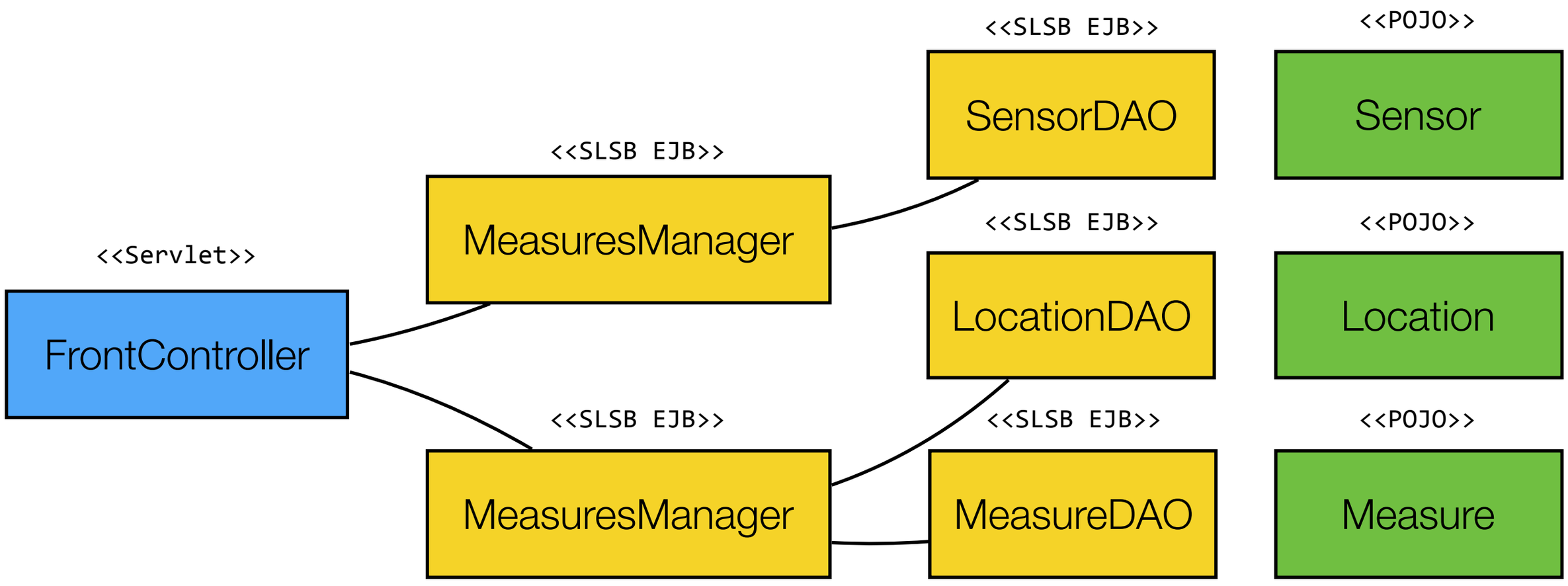
***DAO implementations** handle interactions with specific data stores*





How do I implement the DAO pattern with Java EE technologies?

- There are **different ways to do it**. Some frameworks (e.g. Spring) do that in the web tier (with POJOs).
- If you use **EJBs**, then your architecture is going to look like this:



Is it possible to have **two EJB classes** that implement the **same interface**?

- In the examples so far (and in most cases in practice), we have always created **one local interface** and **one stateless session bean class**.
- If we define the **DAO interface as a local interface** and implement two stateless session beans (JdbcDAO and JpaDAO), then we have an issue:

The **container is unable to resolve this dependency**, because there is more than one implementation. Which one should it choose?

```
public class MyServlet
                extends HttpServlet {
  @EJB
  SensorDAOLocal sensorDAO;
}
```

```
@Local
public interface SensorDAOLocal {
  public long insert(Sensor sensor);
}
```

```
@Stateless
public class SensorJdbcDAO {
          implements SensorDAOLocal
  public long insert(Sensor sensor){}
}
```

```
@Stateless
public class SensorJpaDAO {
          implements SensorDAOLocal
  public long insert(Sensor sensor){}
}
```

Is it possible to have **two EJB classes** that implement the **same interface**?

- We can help the container by giving additional information in the annotation.
- If we define the **DAO interface as a local interface** and implement two stateless session beans (JdbcDAO and JpaDAO), then we have an issue:

The **name**, **beanName** and **mappedName** annotation attributes have different purposes.

```
public class MyServlet
                extends HttpServlet {
  @EJB(beanName="SensorJdbcDAO")
  SensorDAOLocal sensorDAO;
}
```

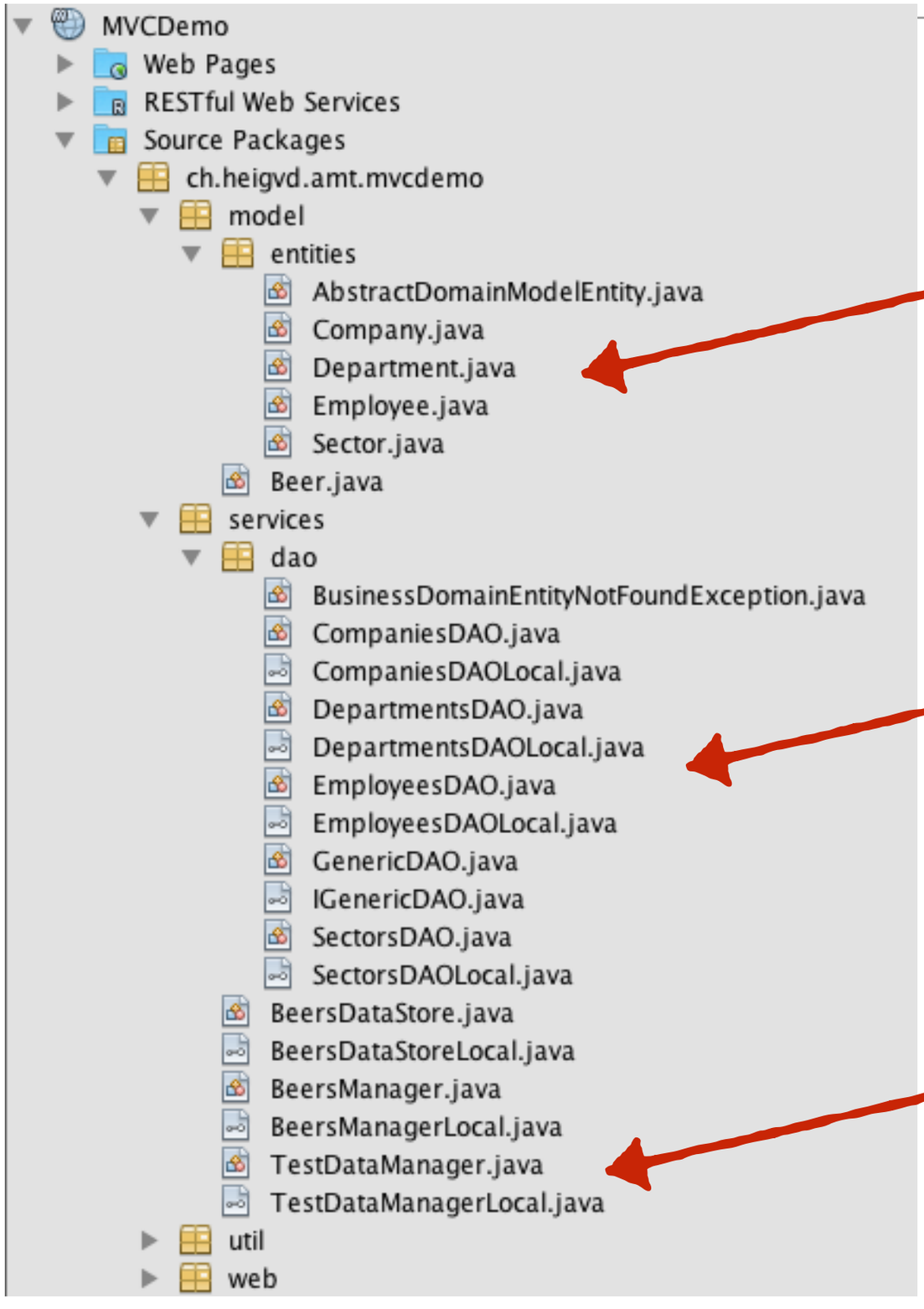
```
@Local
public interface SensorDAOLocal {
  public long insert(Sensor sensor);
}
```

```
@Stateless
public class SensorJdbcDAO {
          implements SensorDAOLocal
  public long insert(Sensor sensor){}
}
```

```
@Stateless
public class SensorJpaDAO {
          implements SensorDAOLocal
  public long insert(Sensor sensor){}
}
```

DAO in the MVCDemo Project

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

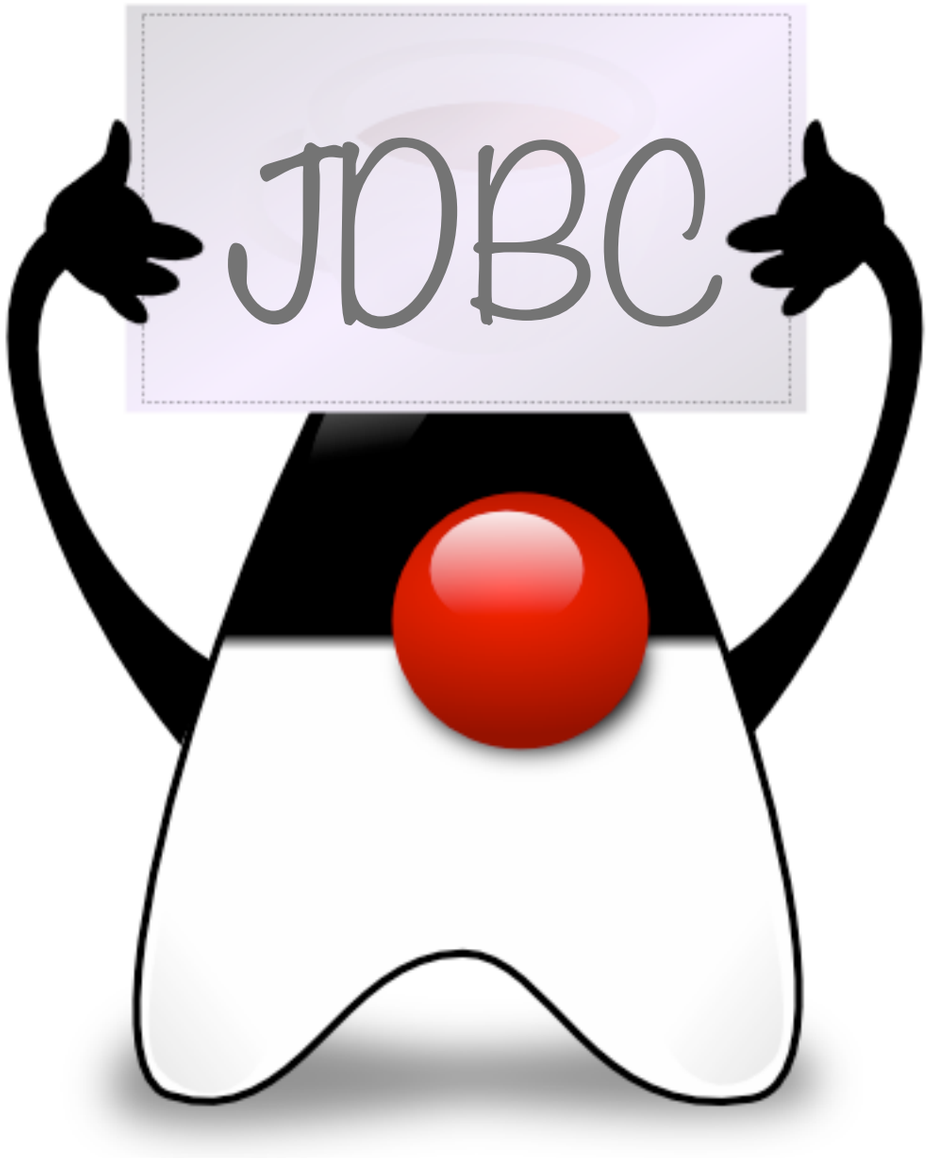


These are the model classes, which can be used with different DAO implementations

These are the DAOs. We only have one implementation, based on JPA.

We use the DAOs to generate test data

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



Java DataBase Connectivity

What is **JDBC**?

- The **Java DataBase Connectivity** is a specification that defines how applications can interact with **relational database** management systems in a **standard way**.
- Its goal is to **create an abstraction layer** between applications and specific RDBMS (MySQL, Oracle, PostgresSQL, DB2, etc.).
- Through this abstraction layer, applications can **submit SQL queries to read**, **insert**, **update** and **delete** records in tables.
- Applications can also get **metadata** about the relational schema (table names, column names, etc.).

What does it look like?

```
@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;
}
```

dependency injection

get a connection from the pool

create and submit a SQL query

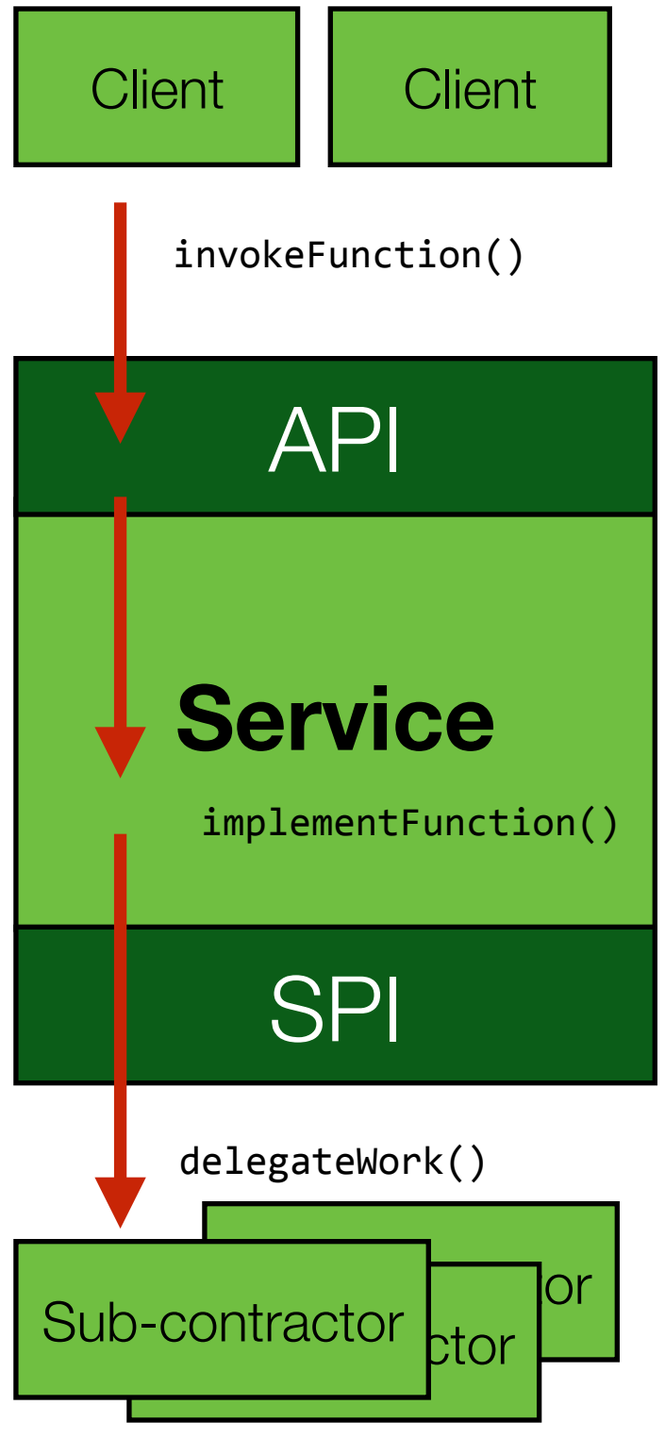
scroll through the tabular result set

get data from the result set

return the connection to the pool

What is the difference between an **API** and a **SPI**?

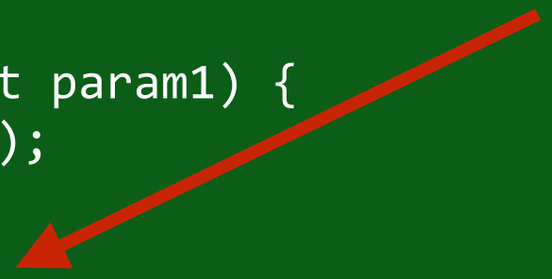
- An **Application Programming Interface** (API) is a **contract** between a client and a service.
- It **defines what the client can request** from the service.
- A **Service Provider API** (SPI) is a contract between a service and its **subcontractors** (components to which it delegates some of the work).
- It **defines what the subcontractors need to do** in order to receive work from the service.



What is the difference between an **API** and a **SPI**?

```
public interface ServiceAPI {
  public void invokeFunction1();
  public String invokeFunction2(Object param1);
}
```

```
public class Service implements ServiceAPI {
  private ServiceSPI provider;

  public void invokeFunction1() { provider.delegateWork(null); };

  public String invokeFunction2(Object param1) {
    doSomething(); delegateOtherWork();
  }

  public void registerServiceProvider(ServiceSPI provider) {
    this.provider = provider
  }
}
```



```
public interface ServiceSPI {
  public void delegateWork(String[] params);
  public void delegateOtherWork();
  public void doSomething();
}
```

In some cases, the SPI is an **extension** of the API.

```
public interface ServiceAPI {
  public void invokeFunction1();
  public String invokeFunction2(Object param1);
}
```

```
public class Service implements ServiceAPI {
  private ServiceSPI provider;

  public void invokeFunction1() { provider.invokeFunction1(); };

  public String invokeFunction2(Object param1) {
    provider.invokeFunction2(param1); doSomethingNotExposedInAPI();
  }

  public void registerServiceProvider(ServiceSPI provider) {
    this.provider = provider
  }
}
```

```
public interface ServiceSPI extends ServiceAPI {
  public void doSomethingNotExposedInAPI();
}
```

What is **JDBC**?

JDBC API

`java[x].sql.*` interfaces

JDBC Service (provided by JRE)

`java[x].sql.*` classes

JDBC SPI (extends JDBC API)

JDBC MySQL driver

implements `java[x].sql.*` interfaces

How is it possible to **obtain a reference** to a JDBC service provider (driver)?

- At some point, the application wants to **obtain a reference to a specific provider**, so that that it can invoke JDBC functions.
- The method depends on the Java environment. You do not the same thing if you are in a **Java SE** or **Java EE** environment.

Java SE

`java.sql.DriverManager`

*Think "**explicit** class loading and connection URLs"*

Java EE

`java.sql.DataSource`

*Think "**managed** resources and "dependency injection"*

How do I **obtain a reference** to a JDBC service provider in **Java SE**?

- In Java SE, the **DriverManager** class addresses this need:
  - It is used by clients who use the API.
  - It is also used by drivers who implement the SPI.
- Think of it as a **broker**, or a **registry**, who puts clients and service providers in relation.
- As a client, I am **explicitly** loading JDBC drivers (1 or more).
- As a client, I am **explicitly** telling with which database I want to interact (via a URL). The URL is used both to find a proper driver and to establish a connection (e.g. hostname, port, etc.).

How do I **obtain a reference** to a JDBC service provider in **Java SE**?

- From the specifications: “Key `DriverManager` methods include:

1. A service provider registers itself in the directory.

  - `registerDriver` — this method **adds a driver to the set of available drivers** and is invoked implicitly when the driver is loaded. The registerDriver method is typically called by the static initializer provided **by each driver**.

Used by **SPI** implementations

  - `getConnection` — the method the **JDBC client** invokes to establish a connection. The invocation includes a JDBC URL, which the `DriverManager` passes to each driver in its list until it finds one whose `Driver.connect` method recognizes the URL. That driver returns a `Connection` object to the `DriverManager`, which in turn passes it to the application.”

Used by **API** clients

2. A client looks for a service provider in the directory.

How do I **obtain a reference** to a JDBC service provider in **Java SE**?

Client

```
Class.forName(“ch.heigdb.HeigDbDriver”);
DriverManager.getConnection(“jdbc:heigdb://localhost:2205”);
```

JDBC Service (provided by JRE)

```
java.sql.DriverManager
registerDriver(Driver driver)
Connection getConnection(String url)
```

JDBC HeigDB driver

```
public class HeigDbDriver implements java.sql.Driver {

    static {
        DriverManager.registerDriver(new SomeDriver());
    }
    public boolean acceptsURL(String url) {};
    public Connection connect(String url, Properties p) {};
}
```

1. Load a class
2. “I am an SPI provider”
3. “Find a SPI provider that will connect me to this DB”
4. “Can you connect me with this DB?”
5. “Connect me with this DB”

How do I **obtain a reference** to a JDBC service provider in **Java EE**?

- In Java EE, the **DataSource** interface is used for managing DB connections.
  - It is used by **application components** (servlets, EJBs, etc.) to obtain a connection to a database.
  - It is also used by **system administrators**, who define the **mapping** between a logical data source name and a concrete database system (by configuration).
- As a developer, I am only using a logical name and I know that it will be bound to a specific system at runtime (but I don't care which…).
- As a developer, I obtain a DataSource either by doing a **JNDI lookup** or via **dependency injection** (with annotations).

How do I **obtain a reference** to a JDBC service provider in **Java EE**?

Client

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/theAppDatabase");
```

OR

3
```
@Resource(lookup="jdbc/theAppDatabase")
DataSource ds;
```

4
```
ds.getConnection();
```

JDBC Service (provided by Java EE)

`java.sql.DataSource`

mysql-connector-java-5.1.33.jar

1 MySQL

Install a **driver** (.jar file) in the app server (/lib/)

2

Create a (logical) data source…

3

… and map it to a (physical) connection pool

Edit JDBC Resource

localhost:4848/common/index.jsf

Home | About... | Help

User: anonymous | **Domain:** domainAMT | **Server:** localhost

GlassFish™ Server Open Source Edition

Tree

- server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Monitoring Data
- Resources
  - Concurrent Resources
  - Connectors
  - JDBC
    - JDBC Resources
      - jdbc/__TimerPool
      - jdbc/__default
      - jdbc/myDataSource
      - jdbc/sample
    - JDBC Connection Pools
      - DerbyPool
      - SamplePool
      - __TimerPool
      - mysql_mysql_rootPool
  - JMS Resources
  - JNDI
  - JavaMail Sessions
  - Resource Adapter Configs
- Configurations

Edit JDBC Resource

Save | Cancel

Edit an existing JDBC data source.

Load Defaults

JNDI Name: jdbc/myDataSource

Pool Name: mysql_mysql_rootPool

Use the JDBC Connection Pools page to create new pools

Deployment Order: 100

Specifies the loading order of the resource at server startup. Lower numbers are loaded first.

Description:

Status: Enabled

Additional Properties (0)

Add Property | Delete Properties

Select	Name	Value	Description
No items found.			

A **JDBC Resource** simply defines a **mapping** between a **logical name** (used in the code) and a **concrete DB connection pool** (hooked to a “physical” DB).

Edit JDBC Connection Pool

localhost:4848/common/index.jsf

Home | About... | Help

User: anonymous | **Domain:** domainAMT | **Server:** localhost

GlassFish™ Server Open Source Edition

Tree

- server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Monitoring Data
- Resources
  - Concurrent Resources
  - Connectors
  - JDBC
    - JDBC Resources
      - jdbc/__TimerPool
      - jdbc/__default
      - jdbc/myDataSource
      - jdbc/sample
    - JDBC Connection Pools
      - DerbyPool
      - SamplePool
      - __TimerPool
      - mysql_mysql_rootPool
  - JMS Resources
  - JNDI
  - JavaMail Sessions
  - Resource Adapter Configs
- Configurations

General | **Advanced** | **Additional Properties**

Edit JDBC Connection Pool Properties

Save Cancel

Modify properties of an existing JDBC connection pool.

Pool Name: mysql_mysql_rootPool

Additional Properties (7)

Add Property | Delete Properties

Select	Name	Value	Description
	URL	jdbc:mysql://localhost:3306/mysql?zeroDateTin	
	driverClass	com.mysql.jdbc.Driver	
	Password	akAUKLJdf!!882_2	
	portNumber	3306	
	databaseName	mysql	
	User	technicalAccount	
	serverName	localhost	

The **connection pool** is configured with “physical” database attributes (host, port, credentials, etc.).

What are some of the key JDBC interfaces and classes?

`DriverManager`

`DataSource`

`XADataSource`

- **`DriverManager`** and **`DataSource`** variations provide a means to obtain a **`Connection`**.
- **`XADataSource`** is used for distributed transactions.
- Once you have a **`Connection`**, you can submit SQL queries to the database.
- The most common way to do that is to create a **`PreparedStatement`** (rather than a **`Statement`**, which is useful for DDL commands).
- The response is either a number (number of rows modified by an UPDATE or DELETE query), or a **`ResultSet`** (which is a tabular data set).
- **`ResultSetMetadata`** is a way to obtain information about the returned data set (column names, etc.).

`Connection`

`PreparedStatement`

`ResultSet`

`ResultSetMetaData`

How do I use these classes in my code?

```
@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;
}
```

dependency injection

get a connection from the pool

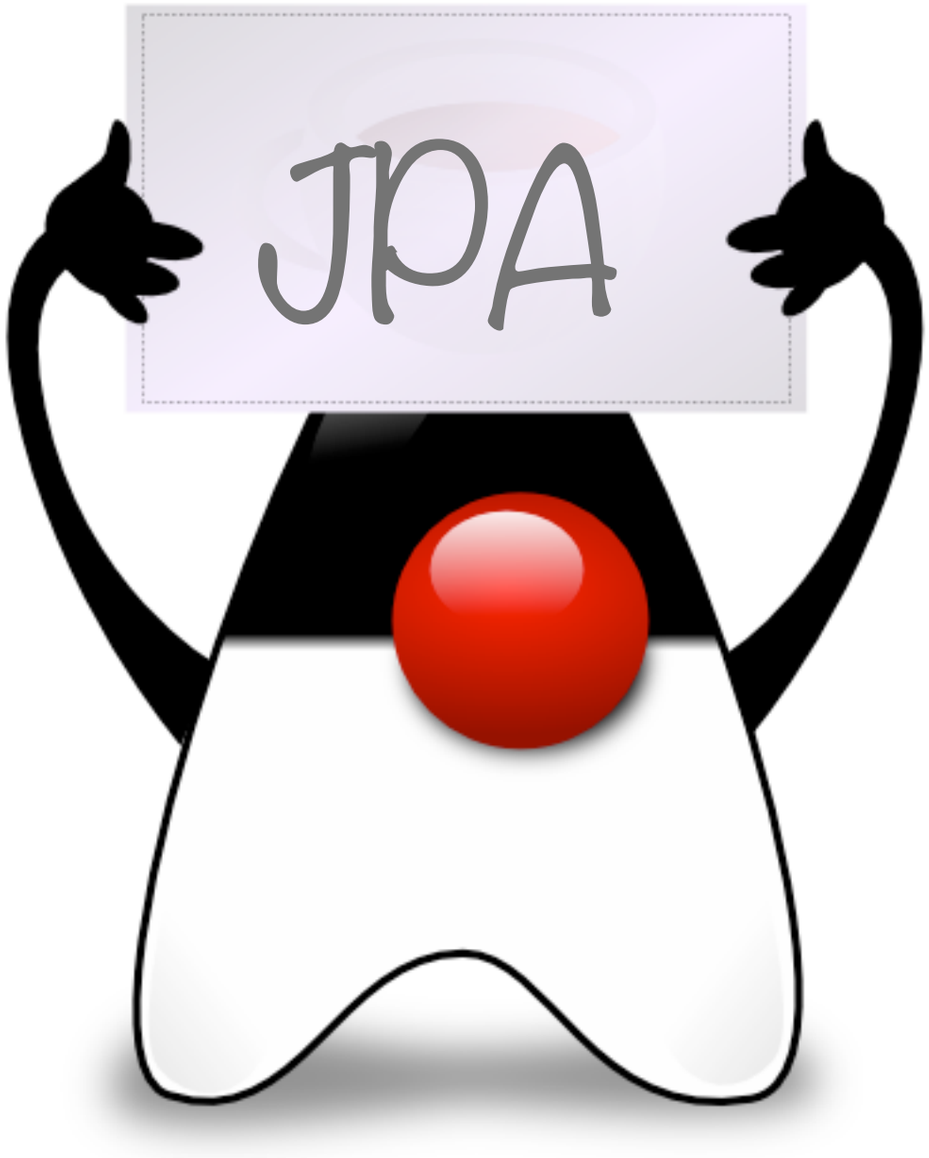
create and submit a SQL query

scroll through the tabular result set

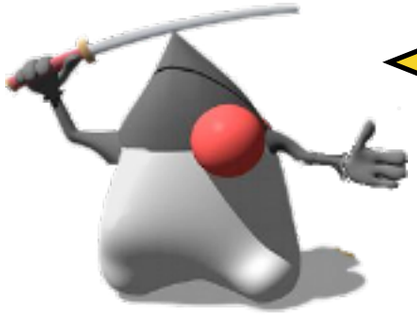
get data from the result set

return the connection to the pool

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



Java Persistence API (JPA)



Foreword

- Today, we will **introduce** the **JPA API** and **related concepts**.
- Some of them will be presented quite quickly, as we do not need them to complete the first JPA lab.
- 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!

<<implementation>>
Business Service

Do a CRUD operation for me!

<<implementation>>
DAOFactory

DAO getDAO();

<<interface>>
DAO

```
long create(T object);
delete(long id);
update(T object);
findById(long);
findAll();
findByXXX(Object k);
findByYYY(Object k);
```

<<implementation>>
JdbcDAO

<<implementation>>
JpaDAO

<<implementation>>
MongoDAO

<<implementation>>
RedisDAO

<<implementation>>
LdapDAO

<<implementation>>
FileSystemDAO

MySQL

PostgreSQL

Oracle

MongoDB

redis

LDAP server

File System

Is it possible to have **two EJB classes** that implement the **same interface**?

- We can help the container by giving additional information in the annotation.
- If we define the **DAO interface as a local interface** and implement two stateless session beans (JdbcDAO and JpaDAO), then we have an issue:

The **name**, **beanName** and **mappedName** annotation attributes have different purposes.

```
public class MyServlet
                extends HttpServlet {
  @EJB(beanName="SensorJpaDAO")
  SensorDAOLocal sensorDAO;
}
```

```
@Local
public interface SensorDAOLocal {
  public long insert(Sensor sensor);
}
```

```
@Stateless
public class SensorJdbcDAO {
          implements SensorDAOLocal
  public long insert(Sensor sensor){}
}
```

```
@Stateless
public class SensorJpaDAO {
          implements SensorDAOLocal
  public long insert(Sensor sensor){}
}
```

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).

Java Persistence API

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

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.

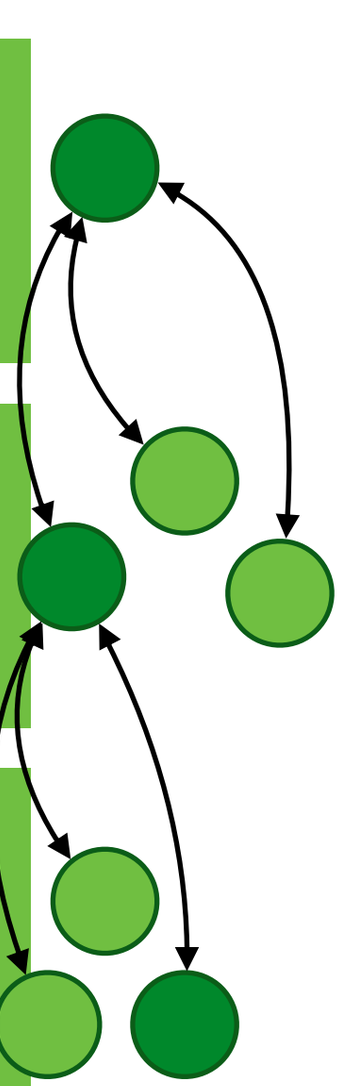
And you let JPA handle the interactions with the database. The schema can be generated automatically, the SQL queries as well.

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

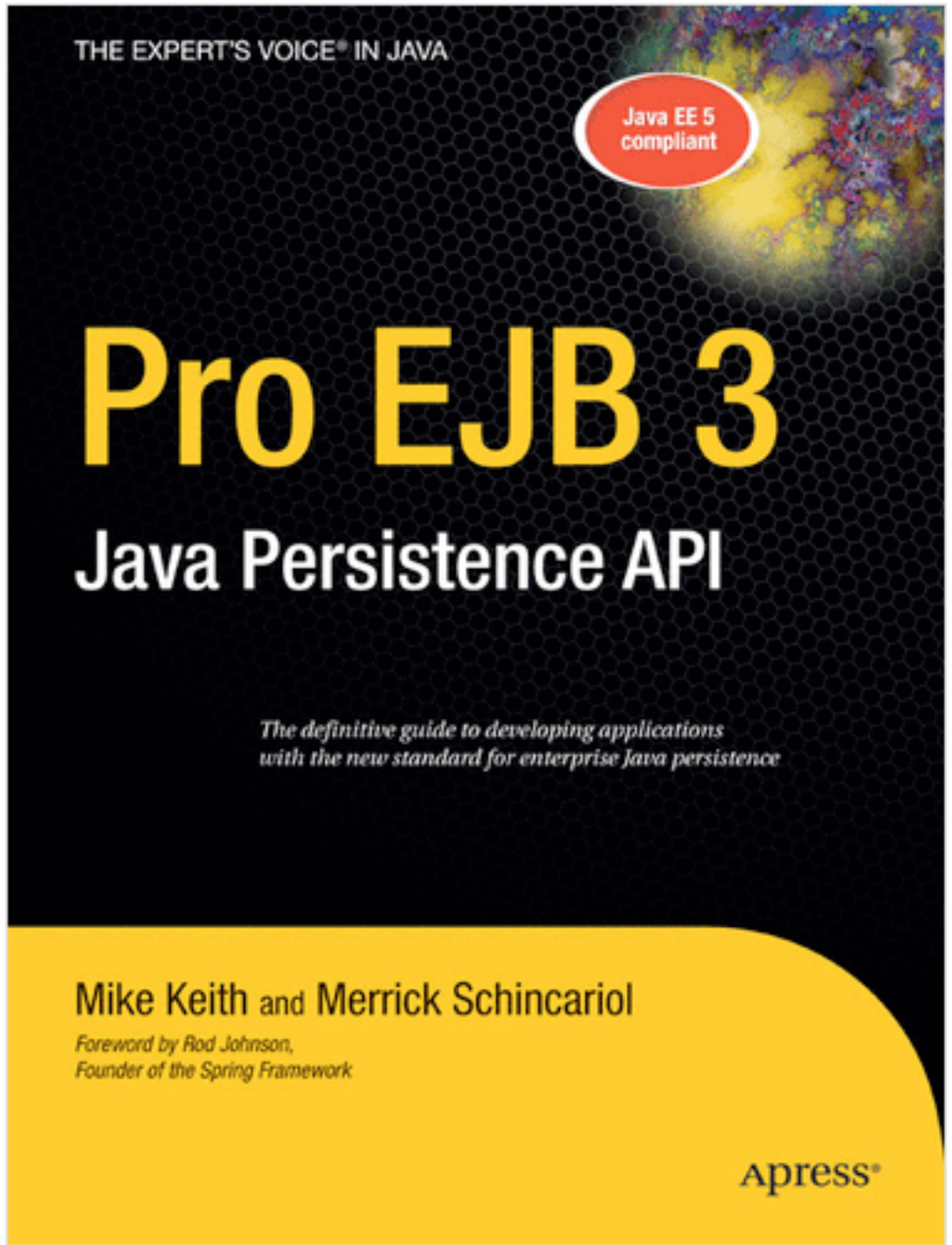
ID	ORDER_ID	ITEM	QTY	U.PRICE
89123	**20983**	989	1	56
89124	**20983**	123	1	100
89125	**20983**	223	4	25

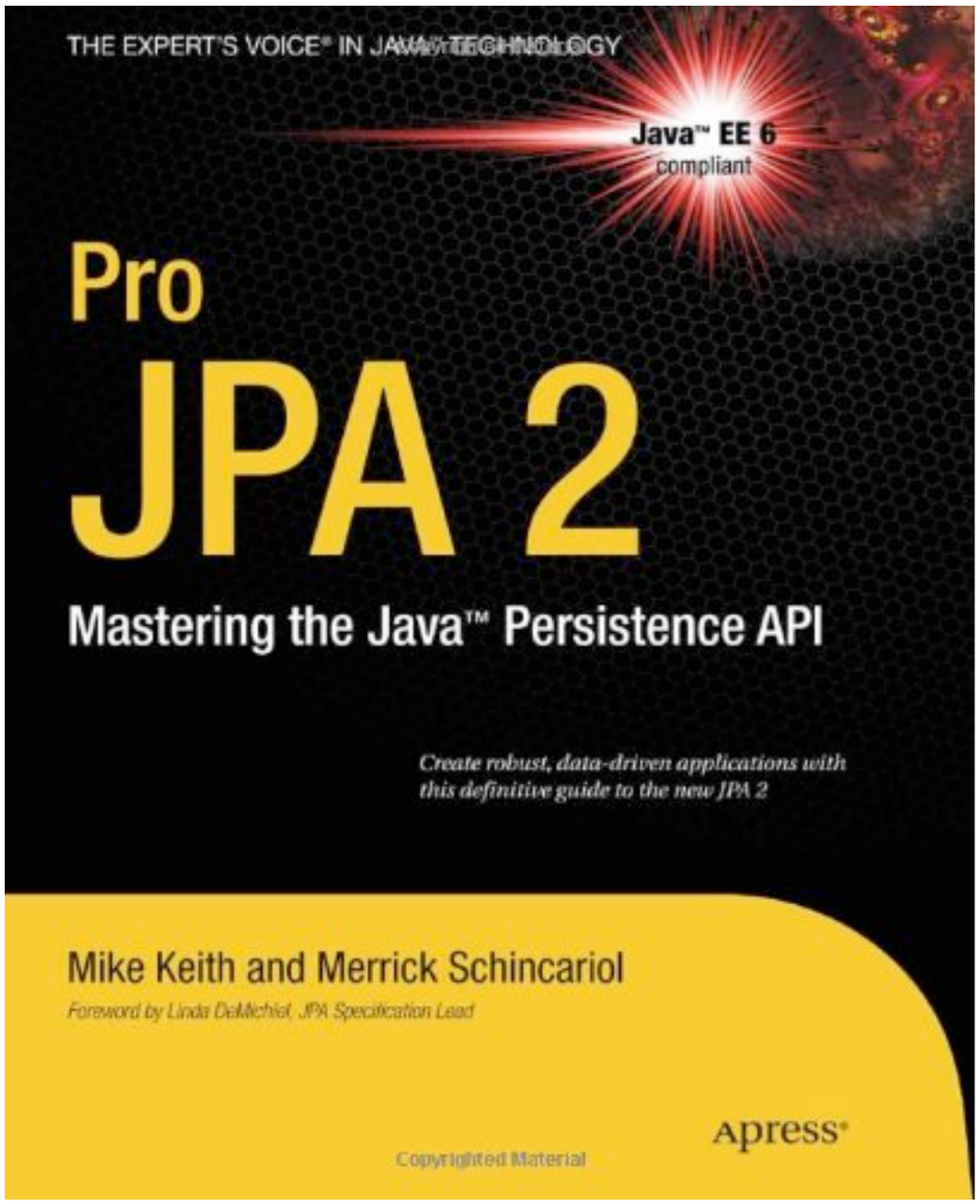
```
Customer customer = CustomersManager.findById(101);
customer.getOrders().get(0).getLines().get(0).getItemRef();
```

```
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
```

The JPA Bible

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





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(...);
```

Java Persistence API

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

*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.

Java Persistence API

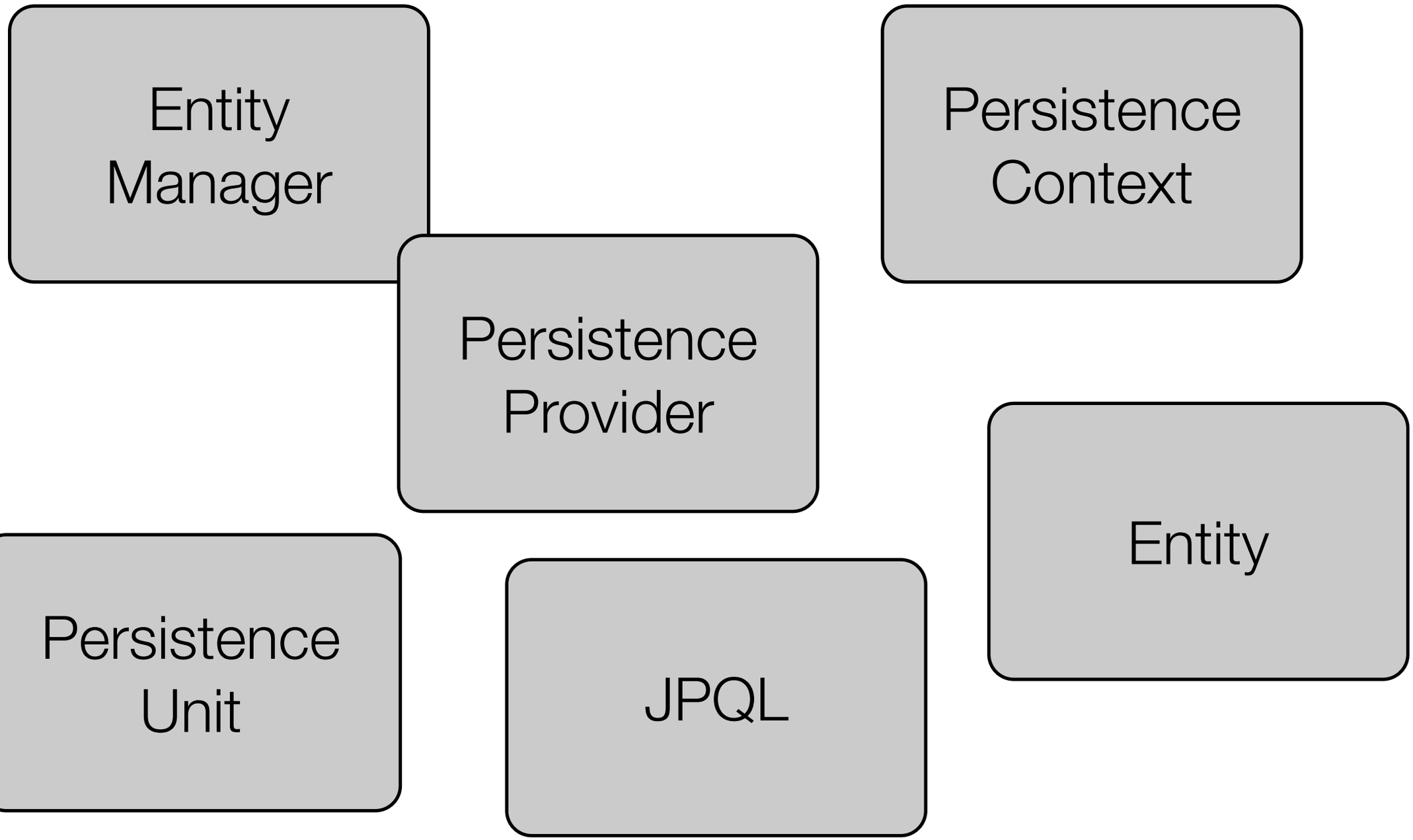
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

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

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



Persistence Provider

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

- 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...

JPA entities

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

- **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

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

- 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

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

- 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

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

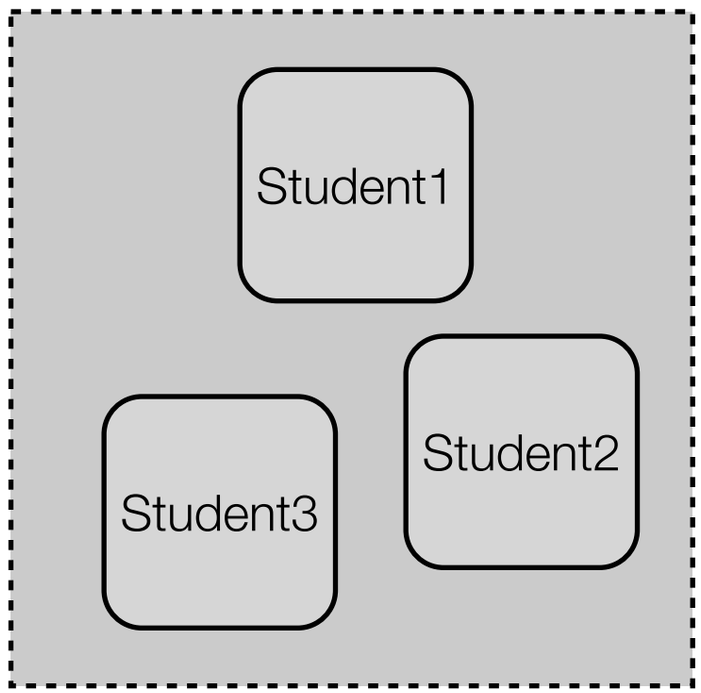
- You can use the Entity Manager in **different types of components**: EJBs, servlets, java applications, etc.
- Using the Entity Manager from **EJBs** is easy. You simply ask the container to inject a reference to the Entity Manager in a variable, with an annotation.
- Using the Entity Manager in the **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

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

- 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

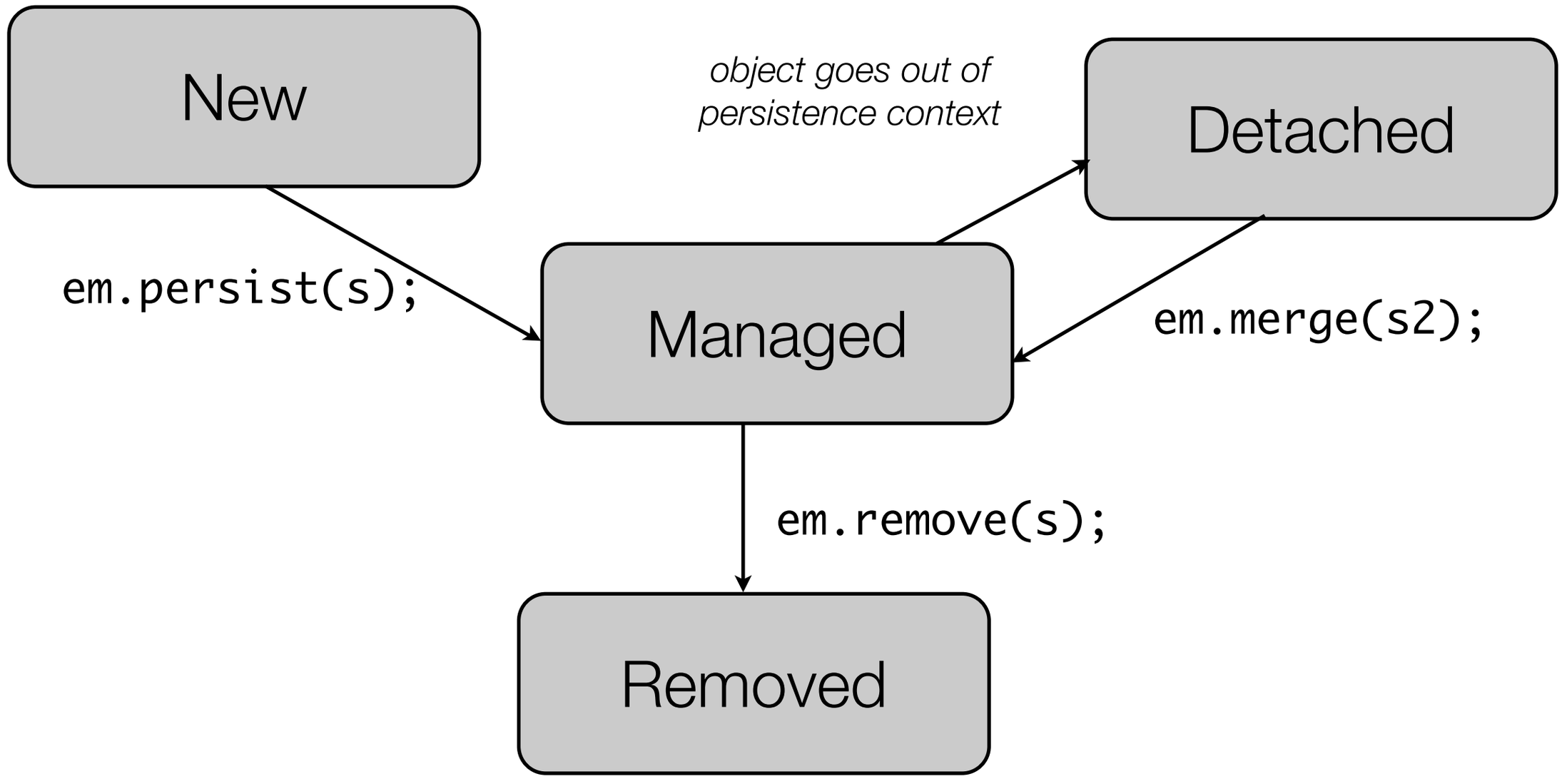
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

- "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

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

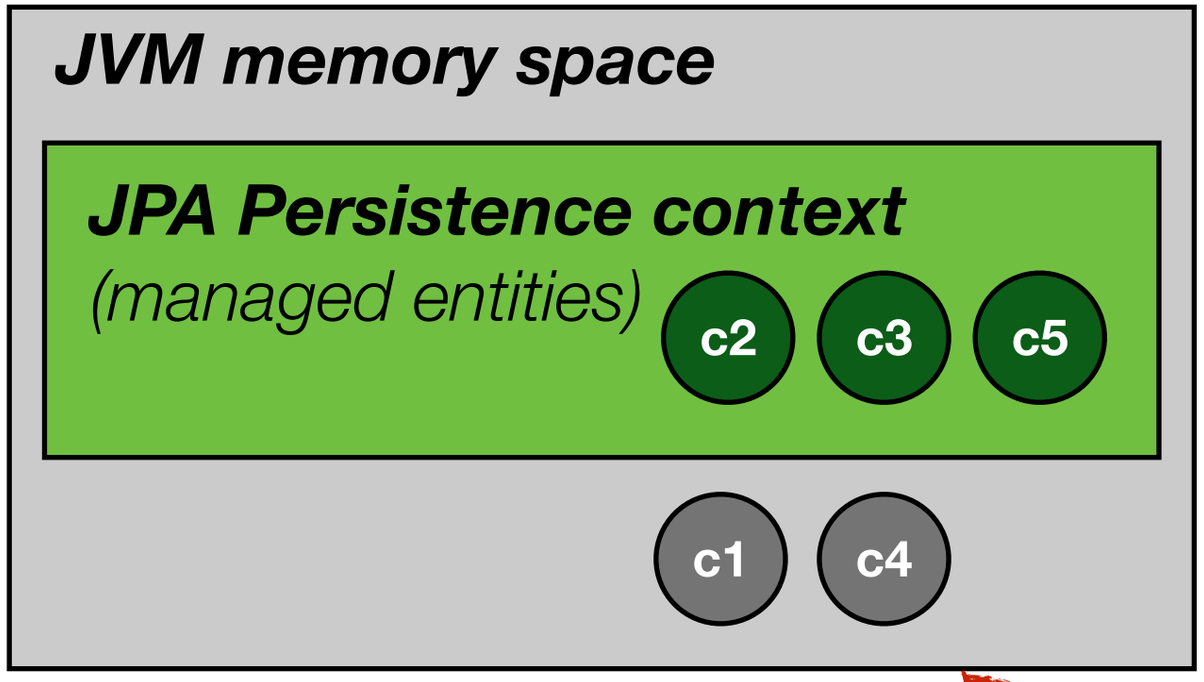
Think what happens when an EJB returns an object to a servlet!



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 rollbacked 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

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

```
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

em.persist(c2);
// c2 is in the 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**").

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

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

- 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

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

- 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.

Entity Relationships

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

```
@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

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

```
@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

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

- 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)

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

- 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)

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

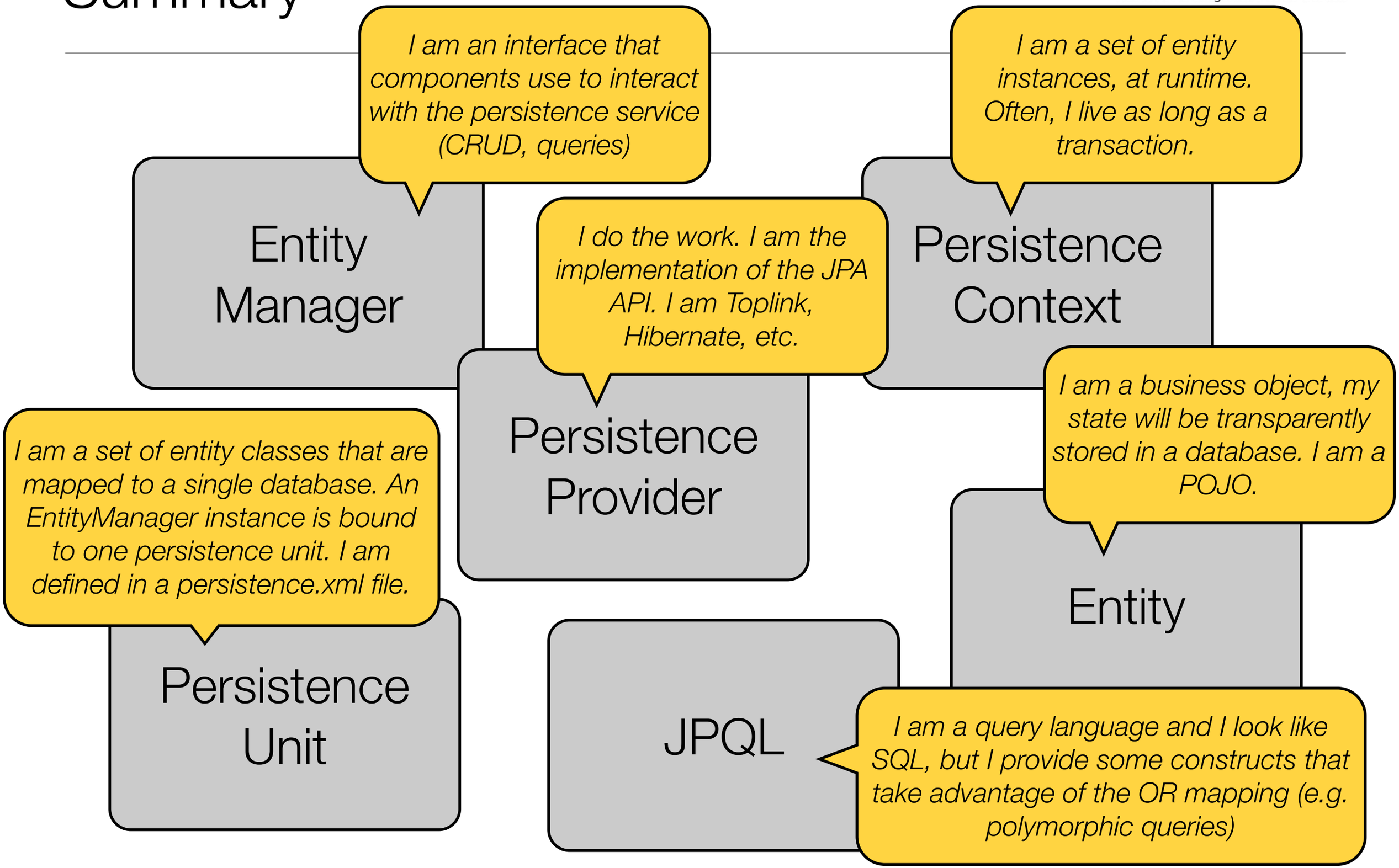
- You can group all your queries at the same place (vs. directly in the service method). Common practice is to use the @NamedQuery in the Entity Class source.

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

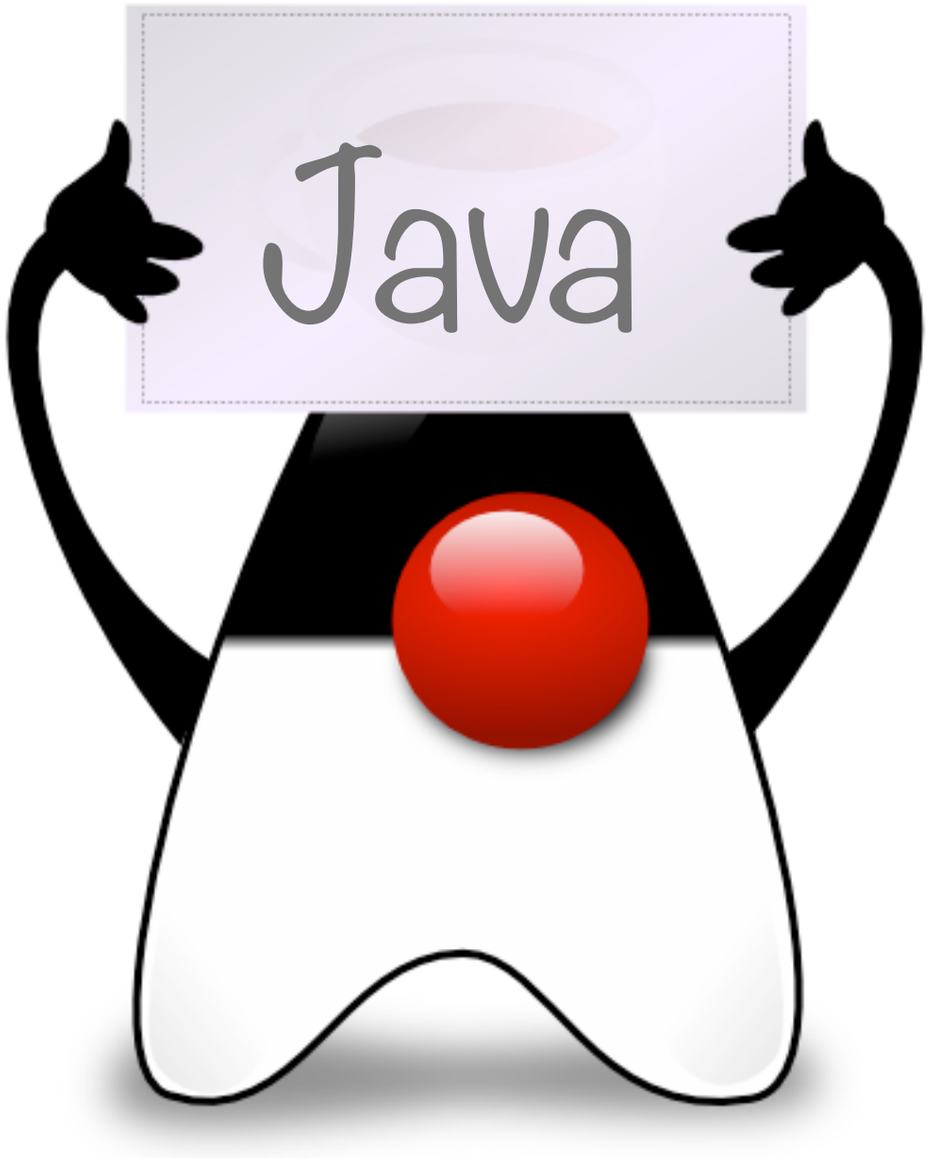
```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
.setParameter("custName", "Smith")
.getResultList();
```

Summary

heig-vd
Haute Ecole d'Ingénierie et de Gestion



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



Java Reflection & JavaBeans

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

2 related questions / observations

Question 1: why do we write this **static block** and isn't it a **dirty hack**?

Client

```
Class.forName("ch.heigdb.HeigDbDriver");
DriverManager.getConnection("jdbc:heigdb://localhost:2205");
```

JDBC Service

`java.sql.DriverManager`

JDBC HeigDB driver

```
public class HeigDbDriver implements java.sql.Driver {

    static {
        DriverManager.registerDriver(new SomeDriver());
    }
    public boolean acceptsURL(String url) {};
    public Connection connect(String url, Properties p) {};
}
```

Why do we do that?

Why don't we write something like:

```
HeigDbDriver driver = new HeigDbDriver();
driver.init();
```

```
public class HeigDbDriver implements java.sql.Driver {

  public void init() {
    DriverManager.registerDriver(new SomeDriver());
  }
  public boolean acceptsURL(String url) {};
  public Connection connect(String url, Properties p) {};
}
```

Question 2: 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**?

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

As a matter of fact, these 2 questions are closely related!

Answering the 1st question will give you a solution for the 2nd!

Let's compare two options carefully:

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

Client

```
Class.forName("ch.heigdb.HeigDbDriver");
DriverManager.getConnection("jdbc:heigdb://localhost:2205");
```

`ch.heigdb.HeigDbDriver` is a **string**

This means that I can **dynamically load** JDBC drivers, **without changing the code** of the client.

I only need to have the drivers **.jar files** in my class path and to **configure** my client.

Why don't we write something like:

```
HeigDbDriver driver = new HeigDbDriver();
driver.init();
```

`HeigDbDriver` is a hard-coded **Java identifier**

This means that if I want to use another JDBC driver, then I need to **change the client code** and **recompile**.

`Class.forName(String name)` is part of the **Reflection API**, which allow us to write dynamic code in Java.

Edit JDBC Connection Pool

localhost:4848/common/index.jsf

Home | About... | Help

User: anonymous | **Domain:** domainAMT | **Server:** localhost

GlassFish™ Server Open Source Edition

Tree

- server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Monitoring Data
- Resources
  - Concurrent Resources
  - Connectors
  - JDBC
    - JDBC Resources
      - jdbc/__TimerPool
      - jdbc/__default
      - jdbc/myDataSource
      - jdbc/sample
    - JDBC Connection Pools
      - DerbyPool
      - SamplePool
      - __TimerPool
      - mysql_mysql_rootPool
  - JMS Resources
  - JNDI
  - JavaMail Sessions
  - Resource Adapter Configs
- Configurations

General | **Advanced** | **Additional Properties**

Edit JDBC Connection Pool Properties

Save Cancel

Modify properties of an existing JDBC connection pool.

Pool Name: mysql_mysql_rootPool

Additional Properties (7)

Add Property | Delete Properties

Select	Name	Value	Description
	URL	jdbc:mysql://localhost:3306/mysql?zeroDateTin	
	driverClass	com.mysql.jdbc.Driver	
	Password	akAUKLJdf!!882_2	
	portNumber	3306	
	databaseName	mysql	
	User	technicalAccount	
	serverName	localhost	

Through this interface, I am **configuring** Glassfish and asking him to do a:

```
Class.forName("com.mysql.jdbc.Driver");
```

I only need to make sure that the mysql .jar files is in Glassfish's **classpath**

```
@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?

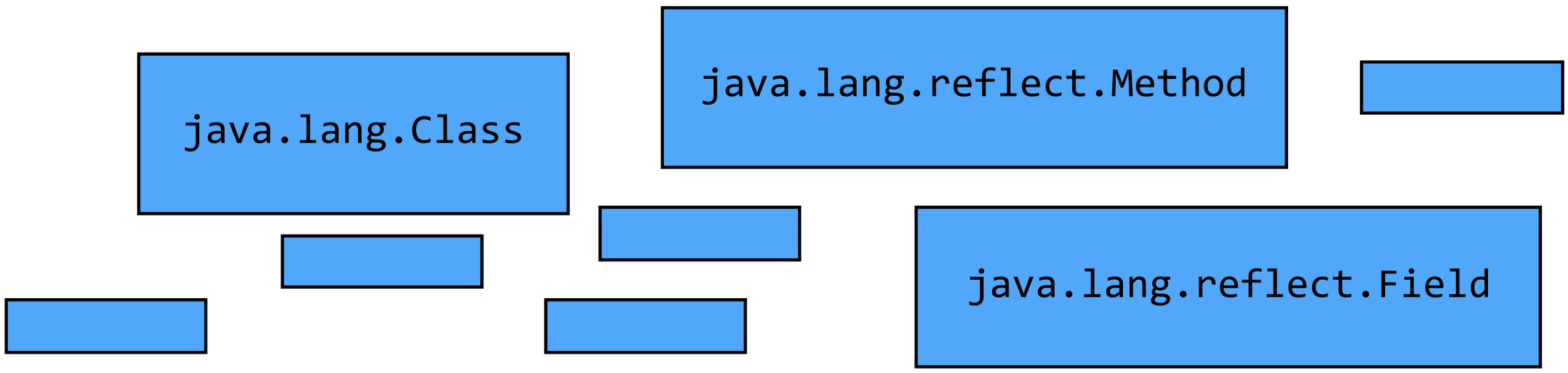
JDBC gives me **metadata** about the DB schema.

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.



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**:

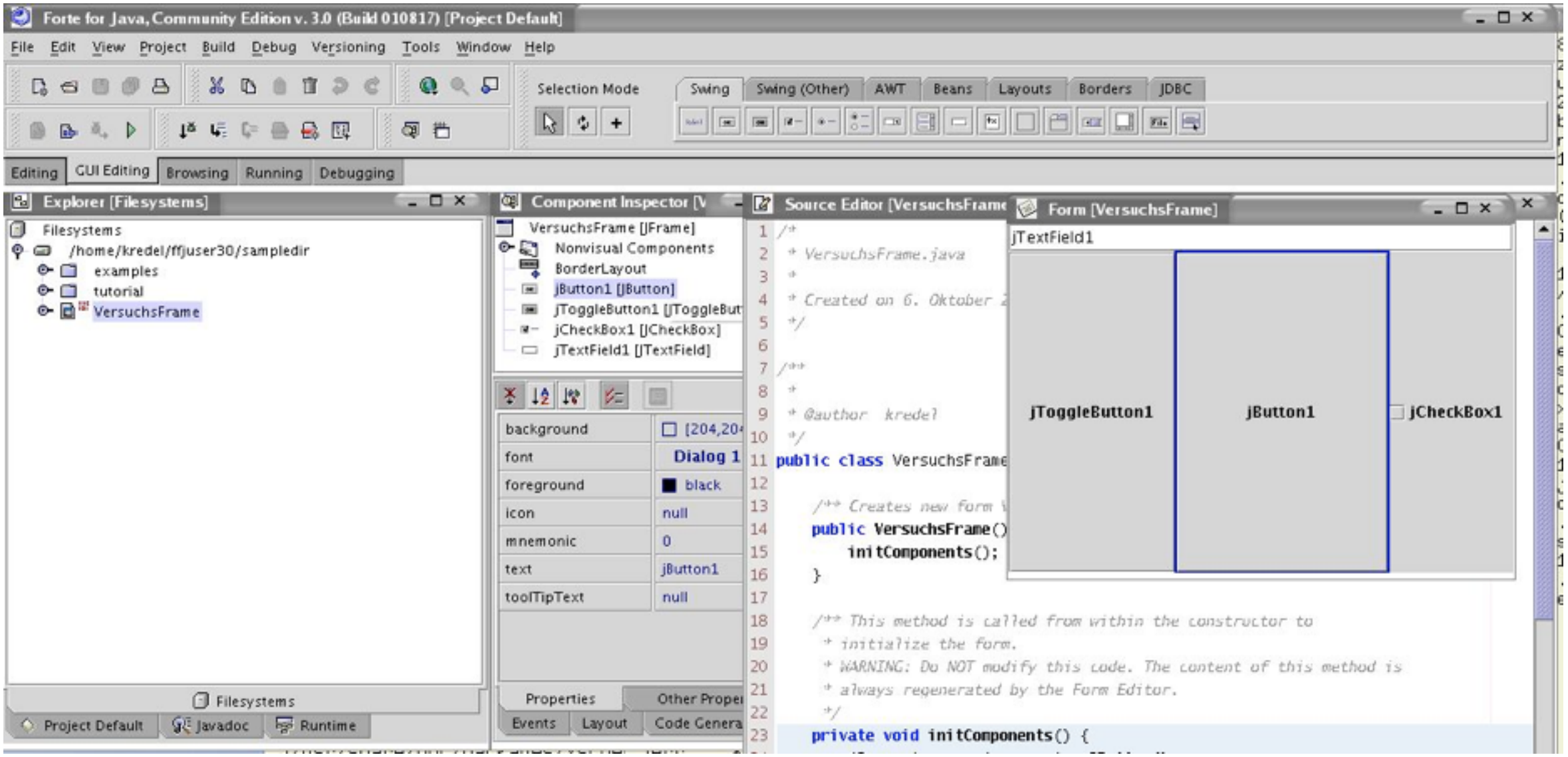
```
Method[] methods = dynamicManagerClass.getMethods();

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**.

```java
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**?

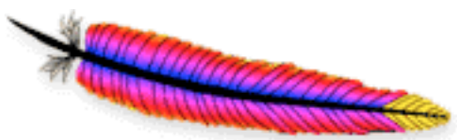
- These **coding and naming conventions** make it easier to **benefit from reflection** in **Java frameworks**:

    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."*



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

commons beanutils™

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

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**.

```
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);
}
```

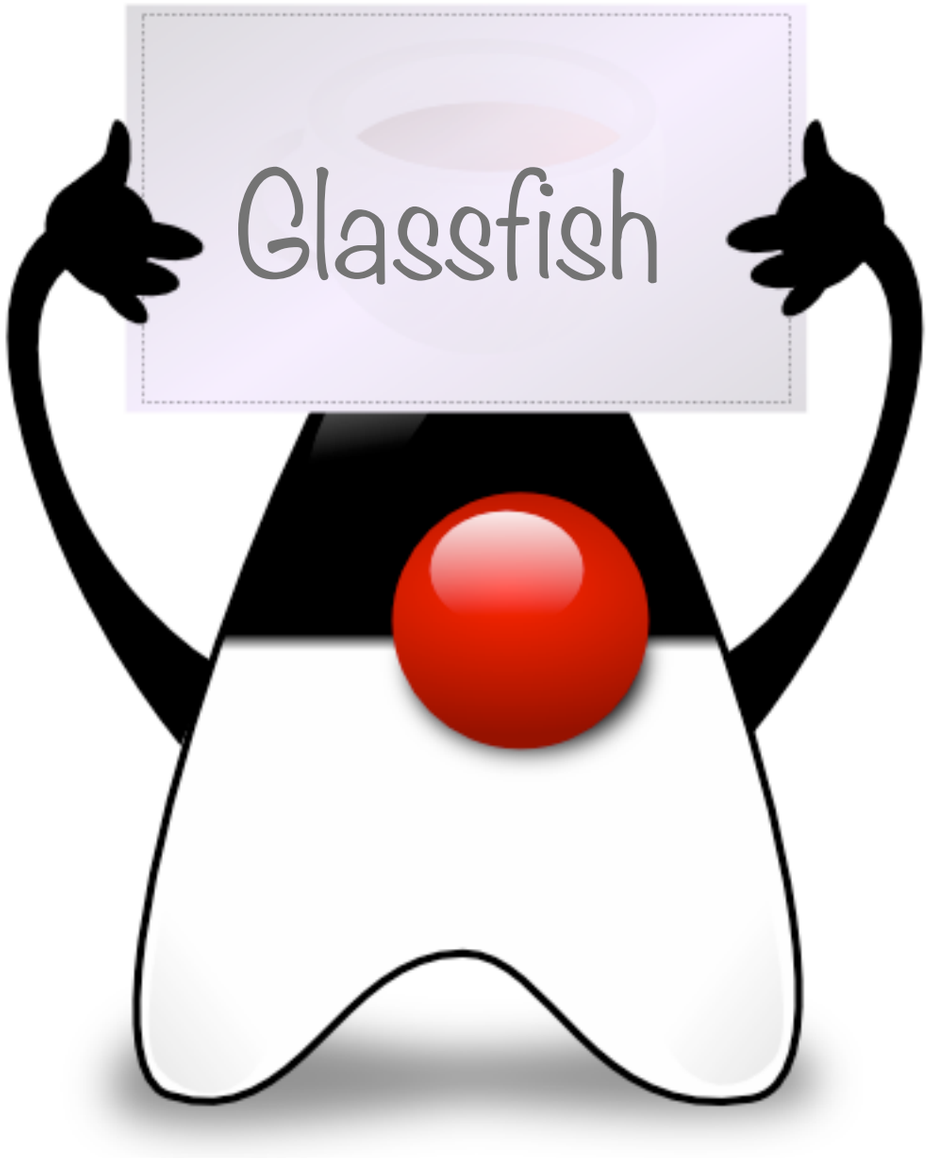
heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

These mechanisms are used by people who build Object Relational Mapping (ORM) frameworks.

We will now look at one of them...

heig-vd

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



Appendix: script the setup of your Glassfish domains

Let's **automate** the creation and configuration of our Glassfish development domain

- Until today, we have used the **default domain (domain1)**, created automagically at installation time.
- Some of you have already had issues with **corrupted domains.** They have used either **Netbeans** or the **asadmin** command line tool to **delete** the domain and **recreate it**.
- During the semester, we will **increasingly automate the build and deployment process** for our Java EE applications. We will start today.

1. We need a **database**! A **clean** database!

- Let's assume that we have installed MySQL on our development machine.
- We want to **create a database** for our application.
- We also want to **create a technical user**, with credentials, to establish communications between our application and MySQL.
- We have to make sure that the technical user has the **permissions** to work with our database.
- We do NOT want to do that manually. **We want to write a script that does that, in an automated and repeatable way**.

2. What do I need to do in **MySQL**?

When you write your script (for instance with **bash**), you will want to use **variables**. This is much better than repeating database or user names throughout the script.

```
DB_NAME=AMTDatabase
DB_TECHNICAL_USER=AMTTechnicalUser
DB_TECHNICAL_USER_PASSWORD=dUke!1400$
```

We can send MySQL commands from the script using separators (<<QUERY_INPUT)

```
mysql -u root -p <<QUERY_INPUT
DROP DATABASE $DB_NAME;
… (other mysql commands…)
QUERY_INPUT
```

In our current setup, we want to **start with a clean, fresh database**. So let's get rid of the previous one (it it exists), before creating a new one.

```
DROP DATABASE $DB_NAME;
CREATE DATABASE $DB_NAME;
```

2. What do I need to do in **MySQL**?

Once our database is created, let's **create a technical user**. MySQL is making a difference between a user who accesses the server from the same machine (localhost) and the same user who accesses the server from a remote machine.

```
DROP USER '$DB_TECHNICAL_USER'@'localhost';
DROP USER ‘$DB_TECHNICAL_USER'@'%';

CREATE USER '$DB_TECHNICAL_USER'@'localhost' IDENTIFIED BY
‘$DB_TECHNICAL_USER_PASSWORD';

CREATE USER '$DB_TECHNICAL_USER'@'%' IDENTIFIED BY ‘$DB_TECHNICAL_USER_PASSWORD';
```

Finally, let's give **permissions** to the technical user to do whatever he wants we our new database.

```
GRANT ALL PRIVILEGES ON $DB_NAME.* TO '$DB_TECHNICAL_USER'@'localhost';
GRANT ALL PRIVILEGES ON $DB_NAME.* TO '$DB_TECHNICAL_USER'@'%';
```

3. What do I need to do in **Glassfish**?

The first thing is to (re)create a fresh domain, from scratch. But since we may already have one, we first need to stop and delete it. Let's use the **asadmin** command for that.

```
DOMAIN_NAME=domainAMT

asadmin stop-domain $DOMAIN_NAME
asadmin delete-domain $DOMAIN_NAME
asadmin create-domain --nopassword=true $DOMAIN_NAME
```

The applications deployed in our domain will want to connect to our MySQL database. For that purpose, we need to **copy the MySQL jdbc driver** into the **lib folder**, under our new **domain folder**.

```
cp mysql-connector-java-5.1.33-bin.jar ../domains/$DOMAIN_NAME/lib
```

3. What do I need to do in **Glassfish**?

Now that we are done with the basic domain setup, we can **start** it.

```
asadmin start-domain $DOMAIN_NAME
```

The last thing that we need for now, is to add a **jdbc connection pool** and a **jdbc resource** in our domain. Of course, **asadmin** provides a way to do that.

```
asadmin create-jdbc-connection-pool \
  --restype=javax.sql.XADataSource \
  --datasourceclassname=com.mysql.jdbc.jdbc2.optional.MysqlXADataSource \
  --property User=$DB_TECHNICAL_USER:Password=
$DB_TECHNICAL_USER_PASSWORD:serverName=localhost:portNumber=3306:databaseName=
$DB_NAME $JDBC_CONNECTION_POOL_NAME
```

```
./asadmin create-jdbc-resource --connectionpoolid $JDBC_CONNECTION_POOL_NAME
$JDBC_JNDI_NAME
```

We can even use asadmin to **ping** our database and validate our setup.

```
asadmin ping-connection-pool $JDBC_CONNECTION_POOL_NAME
```

4. What if I want to **feed the DB**?

In some cases, it's interesting to already **create tables and rows** in the script (makes testing easier during development):

```
USE $DB_NAME;
CREATE TABLE \`sensors\` (
  \`id\` int(11) NOT NULL AUTO_INCREMENT,
  \`description\` tinytext NOT NULL,
  \`type\` tinytext NOT NULL,
  PRIMARY KEY (\`id\`),
  UNIQUE KEY \`id\` (\`id\`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

INSERT INTO \`sensors\` (\`id\`, \`description\`, \`type\`) VALUES (NULL, 'ROOM_1', 'TEMPERATURE');
INSERT INTO \`sensors\` (\`id\`, \`description\`, \`type\`) VALUES (NULL, 'ROOM_2', 'TEMPERATURE');
INSERT INTO \`sensors\` (\`id\`, \`description\`, \`type\`) VALUES (NULL, 'ROOM_31', 'TEMPERATURE');
INSERT INTO \`sensors\` (\`id\`, \`description\`, \`type\`) VALUES (NULL, 'CAR_12', 'SPEED');
INSERT INTO \`sensors\` (\`id\`, \`description\`, \`type\`) VALUES (NULL, 'CAR_99', 'SPEED');
```

To **validate** your script, you should…

- **Run your script several times in a row**, without any error.
- Check the state of the domain in the **web console**: