Data Persistence Layer

Petr Adámek

Content: Persistence I.

- Introduction to data persistence
  - Where to store data
  - How to work with data (Persistence technologies in Java EE)
  - Architecture of data persistence layer
- Introduction to ORM
  - What is ORM
  - Basic Principles
- JPA
  - Introduction
  - Entities in JPA
  - JPA Components
  - Entity Lifecycle

Content: Persistence II

- JPA
  - Advanced Mapping
  - Querying
    - JPQL
    - Criteria API
- Alternatives
  - EJB 2.x
  - JDO
  - JDBC
  - Embedded SQL
  - Spring JDBC
  - iBatis
- Best practices

INTRODUCTION INTO DATA PERSISTENCE LAYER

Where to store data

Data can be stored on different places

- Relational Database (RDBMS)
- Object Database
- XML Database
- DMS (Document Management System)
- CMS (Content Management System)
- Post-relational database (Caché)
  - Temporary
  - Hierarchical
  - Spatial
- Key/Value (No-SQL) Database
- Another Information System (CRM, ERP, etc.)

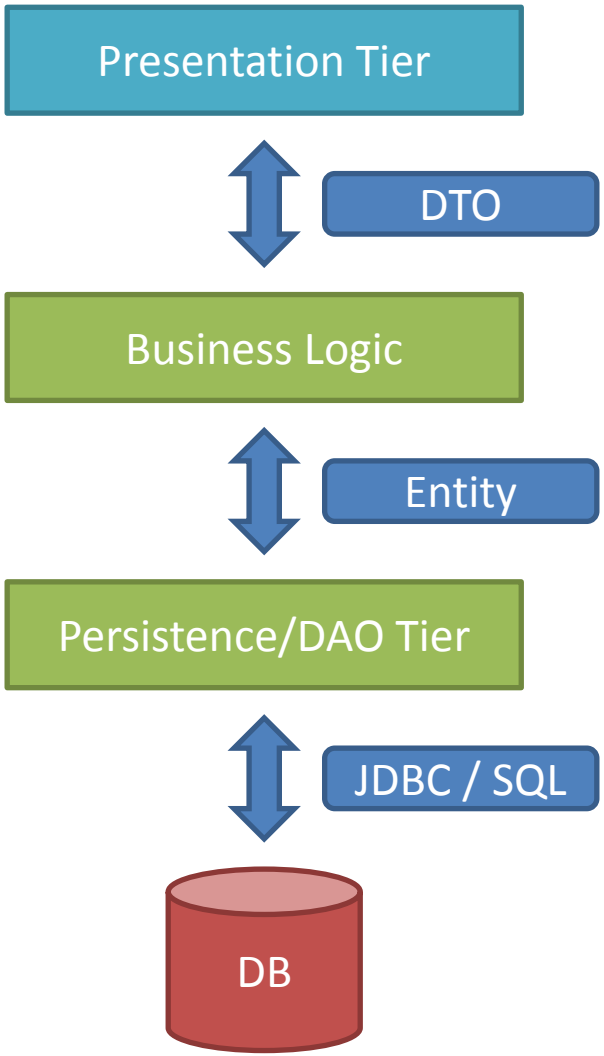
Relational Databases

- The most frequent data storage for enterprise applications
  - Relational data model is simple but very powerful
  - Suitable and sufficient for most of common applications
  - Good theoretical model (Relational Algebra, Relational Calculus)
  - Simplicity => High Performance (eg. due simple optimizations)
  - Proven and well established technology (40 years of development, tools, standards, reliability, high penetration, lots of experts, etc.)
  - Data are separated from application and can be easily shared between different applications
  - Independent on concrete platform or programming language

Persistence Technologies

- With Relational Model
  - JDBC (low-level API, cumbersome for direct use)
  - Commons DB Utils
  - Spring JDBC
  - iBatis/MyBatis
  - Embedded SQL
- With Object Model (ORM or other model conversion)
  - Legacy EJB 2.x
  - Hibernate
  - JPA
  - JDO

Architecture of persistence layer



Architecture of persistence layer

- DAO (*Data Access Object*) design pattern
  - Separation persistence and data access from business logic
  - Allows to modify persistence implementation or replace persistence technology at all without affecting application logic.
- DTO *(Data Transfer Object) design pattern*
  - Business logic API is independent of entities
  - Data model can be changed without affecting Business logic API and/or presentation tier
  - DTO granularity is independent on entity size (single DTO could contain only subset of entity attributes or could agregate information from mulitple entities)

Transaction management

- Transaction are not controlled on DAO level
- Why?
  - Single transaction can contain more operations provided by different DAO components
  - Transaction management should be independent on persistent technology
- Transaction management will be discussed later

INTRODUCTION TO ORM

What is ORM

Object-relational mapping (ORM)

- Technique for automatic conversion between object model and relational data model
- You work with objects, but they are stored in a traditional relational database.

INSERT INTO people (id, name) VALUES (1, "Pepa");

```
Person p = new Person(1, "Pepa");
em.persist(p);
em.getTransaction().commit();
```

UPDATE people SET name = "Honza" WHERE id = 2;

```
Person p = em.find(Person.class,2);
p.setName("Honza");
em.getTransaction().commit();
```

Why ORM

- As we already discussed, the most frequent storage is RDBMS
- But we usually want to work with Object Model
- Why Object Model?
  - It is natural for object oriented programming language
  - Working with Object Data Model is straightforward, friendly and easy
  - See example

Why ORM: JDBC example

```
public String getPersonName(long personId) throws SQLException {
  PreparedStatement st = null;
  try {
    st = connection.prepareStatement(
      "SELECT name FROM people WHERE id = ?");
    st.setLong(1, personId);
    ResultSet rs = st.executeQuery();
    if (rs.next()) {
      String result = rs.getString("name");
      assert !rs.next();
      return result;
    } else {
      return null;
    }
  } finally {
    if (st != null) { st.close(); }
  }
}
```

Why ORM: JPA Example

```
public String getPersonName(long personId) {
    Person p = em.find(Person.class,personId);
    return p.getName();
}
```

ORM: What We Get and Lost

- ORM Benefits
  - Possibility to work with natural object model
  - Portability between different RDBMS with different SQL Dialect
  - Type checking at compile phase
  - No runtime error in SQL statements
  - Simplifies testing
  - Simpler and clearer code
  - More effective development (auto complete, access to JavaDoc, etc.)

- ORM drawbacks
  - Less control over SQL statements sent to database
  - Less performance in some cases (ORM has some overhead).
  - No access to advantages of relational model and features of RDBMS (e.g. storage procedures)

Basic terms

What we should already know

- **JDBC, SQL, Transaction**

New terms

- **Entity** – domain object representing data stored into database (eg. Person, Invoice, Course).
- **DTO** (*Data Transfer Object*) – object for data encapsulation and transfering between components.
- **POJO** (*Plain Old Java Object*) – simple class without any special requirements. It may not implement any special interface, it may not extend any given class, it may not dependent on any other class, package or framework.

Standards and Approaches

Entity EJB (EJB 2.1/JSR 153; J2EE 1.4)

- Application server with EJB container required.
- Entity is heavyweight component, instances are located in EJB container and accessed through remote calls
- Problem with latences (reason for introducing DTO and DAO patterns).
- CMP or BMP
- JPA is preferred since EJB 3.0

JDO (JDO 3.0/JSR 243)

- General and universal standard for data persistence in Java.
- Not limited to RDBMS, arbitrary storage can be used for storing objects

JPA (JPA 2.0/JSR 317; Java EE 6)

- Java EE standard for ORM (inspired with Hibernate)
- Entity is lightweight POJO which can be freely passed between components, locally or remotely.

EJB 2.x CMP Entity

```
public abstract class PersonBean implements EntityBean {
    private EntityContext context;

    public abstract Long getId();
    public abstract void setId(Long id);
    public abstract String getName();
    public abstract void setName(String name);

    public Long ejbCreate (Long id, String name) throws CreateException {
       setId(id); setName(name); return id;
    }
    public void ejbPostCreate (Long id, String name) throws CreateException {}

    public void setEntityContext(EntityContext ctx)  { context = ctx; }
    public void unsetEntityContext() { context = null; }

    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}

    public PersonDTO getPersonDTO() {
        return new PersonDTO(getId(),getName());
    }
}
```

POJO Entity

```
public class Person {

    private Long id;
    private String name;

    public Long getId()                { return id; }
    public void setId(Long id)         { this.id = id; }
    public String getName()            { return name; }
    public void setName(String name)   { this.name = name; }

    public boolean equals(Object o) {
        if (this == o) { return true; }
        if (getId() == null) { return false; }
        if (o instanceof Person) {
            return getId().equals(((Person) o).getId());
        } else {
            return false;
        }
    }

    public int hashCode() { return id==null?0:id.hashCode(); }
}
```

Mapping Definition

With annotations

- Object model definition and its mapping are on the same place.
- Clear and straightforward
- Easier development and maintenance

With external file (usualy XML)

- Entities are independant on particular ORM technology
- Mapping could be changed without modification of code

With special JavaDoc comments

- For Java 1.4 and older (without annotations support)
- See XDoclet.

Mapping and DB schema definition

Generování schématu databáze na základě definice mapování

- Máme vytvořené entity a definici mapování a chceme si ušetřit práci s vytvářením schématu databáze.
- Je možné automaticky vytvářet tabulky při prvním spuštění aplikace.
- Výhodné zejména při vývoji, kdy dochází ke změnám datového modelu.
- Vhodné, pokud je datový model zcela pod kontrolou naší aplikace.
- Problém, pokud se mění datový model a již máme v databázi existující data.

Generování entit a definice mapování na základě schématu databáze

- Máme vytvořené schéma databáze a chceme si ušetřit práci s vytvářením entit a definicí mapování (např. vyvíjíme aplikaci pro přístup k již existujícím datům).
- Obvykle je nutné vygenerované soubory ručně opravit.

JAVA PERSISTENCE API

JPA Introduction

Java Persistence API

- POJO Entities, inspired by ORM tool Hibernate
- API implemented by various ORM tools from different vendors.
- Just basic functionality, implementations could provide other features and functions through its proprietary API

Versions and specifications

- **JPA 1.0** – part of Java EE 5; created as part of EJB 3.0 (JSR 220), but independent.
- **JPA 2.0** – součást Java EE 6; JSR 317
- **JPA 2.1** – part of Java EE 7; JSR 338

ORM tools implementing JPA

- Hibernate, Open JPA
- TopLink, TopLink Essentials, Eclipse Link

Entity in JPA

Entity

- Represent domain object
- Simple POJO class
- Attributes represents domain object properties
- Attributes accessible with set/get methods
- Mandatory parameterless constructor
- It is useful (but not mandatory) to implement Serializable

Mapping definition

- With annotations or xml file
- *Convention-over-configuration* principle.

JPA Entity

```
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    public Long getId()               { return id; }
    public void setId(Long id)        { this.id = id; }
    public String getName()           { return name; }
    public void setName(String name) { this.name = name; }

    public boolean equals(Object o) {
        if (this == o) { return true; }
        if (getId() == null) { return false; }
        if (o instanceof Person) {
            return getId().equals(((Person) o).getId());
        } else {
            return false;
        }
    }

    public int hashCode() { return id==null?0:id.hashCode(); }
}
```

Example

- Working With Entity

Configuration

Configuration

- Stored in persistence.xml
- Contains one or more *Persistence Units*.

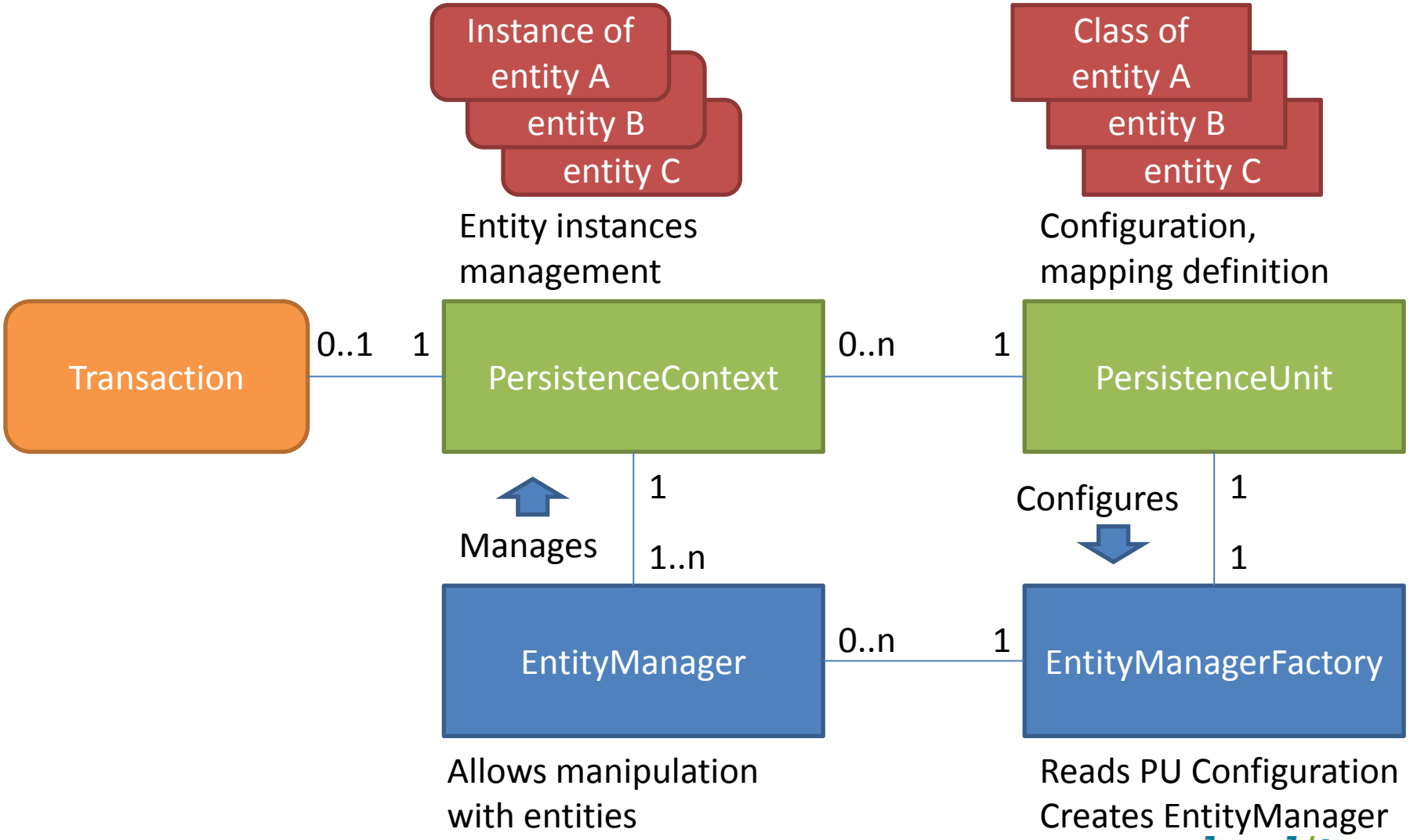
Persistence Unit

- List of classes managed by given Persistence Unit
- Database connection configuration
  - JNDI name of DataSource
  - JDBC url, name, password
- Transaction control configuration (RESOURCE_LOCAL or JTA)
- Table creation strategy

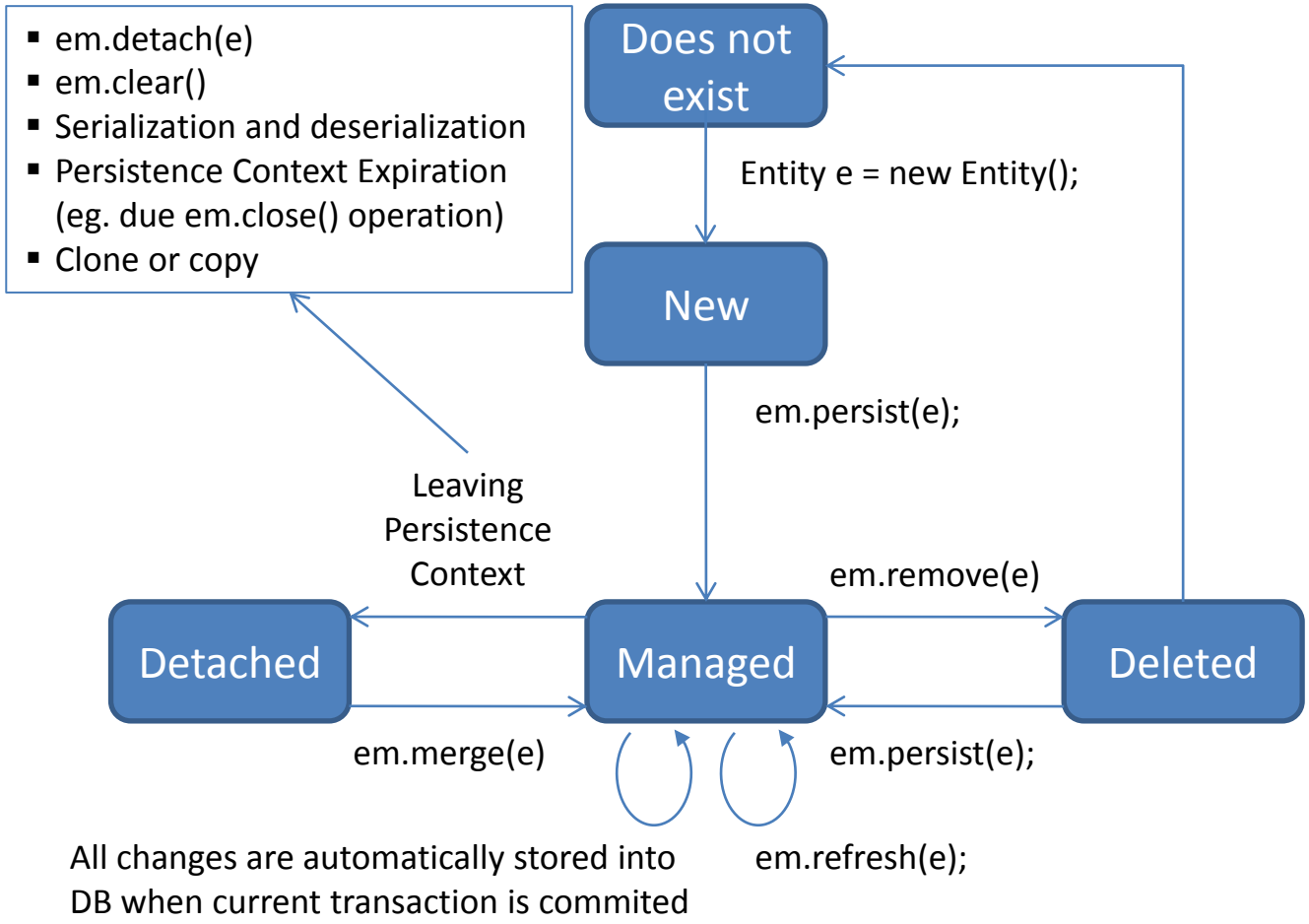
Configuration parameter names

- Vendor specific for JPA 1.0, standardized for JPA 2.0
- Parameters could be also set when creating EntityManagerFactory or EntityManager

JPA Architecture



Entity Lifecycle



Quiz

- Entity Lifecycle

MAPPING DEFINITION

Supported types

- Primitive types: byte, int, short, long, boolean, char, float, double
- Wrapper types: Byte, Integer, Short, Long, Boolean, Character, Float, Double
- Byte and character array types: byte[], Byte[], char[], Character[]
- Large numeric types: java.math.BigInteger, java.math.BigDecimal
- Strings: java.lang.String
- Java a JDBC temporal types: java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp
- Enumerated types: Any system or user-defined enumerated type
- Serializable objects: Any system or user-defined serializable type
- Any object with appropriate convertor (JPA 2.1)

Field mapping configuration

- Data Acces
  - Field access
  - Property access
  - Mixed access (@Access, JPA 2.0)
- @Column
- @Basic
- @Lob
- @Transient or transient

Primary key

- Supported types
  - byte, int, short, long, char
  - Byte, Integer, Short, Long, Character,
  - java.lang.String , java.math.BigInteger
  - java.util.Date, java.sql.Date
  - Floating point types supported but strongly not recommended
- ID generation
  - Sequence, identity, table, Auto
- Composite primary key
  - Key class required

Field mapping

- Data Acces
  - Field access
  - Property access
  - Mixed access (@Access, JPA 2.0)
- Enums
- Temporal types (Date, Time)
- Boolean

Relationships

- Directionality
  - Uniderctional
  - Bidirectional
- Cardinality
  - OneToOne
  - OneToMany
  - ManyToOne
  - ManyToMany
- Owning side
  - @JoinColumn, mappedBy
- Cascade Operations

Lazy fetching

- @Basic, @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
- Useful, but use with care
  - Just performance hint, not necessary supported
  - Problem with detached entities
  - Not universal solution for dependency problem

Embedded object

- @Embeddable, @Embedded
- @AttributeOverrides, @AttributeOverride
- Multiple occurences of the same embedded class in the entity is not allowed in JPA 1.0

Element Collections

- In addition to relationships, also collection of these objects are supported (since JPA 2.0)
  - Simple types
  - Embeddable classes
- @ElementCollection, @CollectionTable

Collection Types

- Set, Collection
  - No aditional information required
- List
  - @OrderBy
  - @OrderColumn (since JPA 2.0)
- Map
  - In JPA 1.0, Map is allowed only in relationships (Value is entity and key is attribute of given entity)
  - In JPA 2.0, any SimpleType, Collection or Embeddable can be key or value
  - @MapKeyColumn, @MapKeyEnumerated

Inheritance

- @MappedSuperclass
- Entity inheritance
  - Single table strategy
  - Multiple table strategy
  - Join table strategy

QUERYING

Querying

- JPQL
  - Query language similar to SQL
  - Supports scalar values, tuples, entities or constructed objects
- Criteria API
  - From JPA 2.0
  - Allows to build query programatically
- Native queries
  - Queries in SQL, non portable, not integrated with JPA infrastructure
  - Use only in exceptional cases

Calling JPQL

Example

- JPQL
  - Executing simple query
  - Named queries
  - Returning scalar values
  - Returning tuples
  - Constructing objects
- Criteria API
  - Simple example

JPQL Examples

OTHER TECHNOLOGIES

EJB 2.x

- Incompatible with DAO Design Pattern
  - Actually, DAO Pattern was designed as replacement for EJB 2.x Entities
- Requires Java EE Application server with EJB Container
- Entity is heavyweight component, instances are located in EJB Container and accessed remotely
- Problem with latencies (reason for introducing DAO and DTO design patterns)
- CMP versus BMP
- JPA is preferred from EJB 3.0

EJB 2.x Entity

```
public abstract class PersonBean implements EntityBean {
    private EntityContext context;

    public abstract Long getId();
    public abstract void setId(Long id);
    public abstract String getName();
    public abstract void setName(String name);

    public Long ejbCreate (Long id, String name) throws CreateException {
       setId(id); setName(name); return id;
    }
    public void ejbPostCreate (Long id, String name) throws CreateException {}

    public void setEntityContext(EntityContext ctx)  { context = ctx; }
    public void unsetEntityContext() { context = null; }

    public void ejbRemove() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}

    public PersonDTO getPersonDTO() {
        return new PersonDTO(getId(),getName());
    }
}
```

Embedded SQL

- SQL written directly in the code
- Code is processed with special preprocessor before compiling
- Preprocessor process SQL expressions, checks their validity, performs type checking a translates them into expression of used programming language.
- Preprocessor requires database connection

```
public String getPersonName(long personId) {
    String name;
    #sql {
        SELECT name INTO :name
        FROM people WHERE id = :personId
    };
    return name;
}
```

Spring JDBC

- Spring library implementing *Template Method* design pattern
- Cleaner code, faster development, easier maintenance
- Unlike ORM or Embedded SQL does not solve the problem with errors in SQL expressions, that become apparent until runtime

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

public String getPersonName(long personId) {
  return jdbcTemplate.queryForObject(
    "SELECT name FROM people WHERE id = ?",
    String.class,personId);
}
```

Apache Commons DBUtils

- Another library implementing *Template Method* design pattern.

```
QueryRunner queryRunner =
  new QueryRunner(dataSource);
ResultSetHandler<String> stringHandler =
  new ScalarHandler<String>();

public String getPersonName(long personId) {
  return queryRunner.query(
    "SELECT name FROM people WHERE id = ?",
    stringHandler,personId);
}
```

iBatis/MyBatis

- Originally Apache project, retired, currently developed as MyBatis
- SQL Queries are separated from code
  - In XML file
  - In annotations (in new versions)
- More powerful that simple libraries like Spring JDBC, more lightweight than ORM

TRANSACTION MANAGEMENT

Transactions

- Rollback in JPA and persistence context
- Transaction scoped Persistence Context

Questions

?