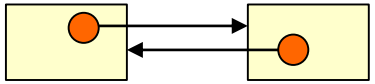
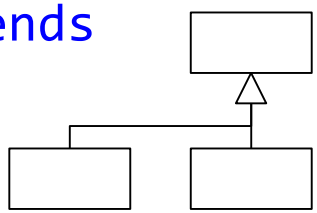
Java Persistence API

- **O/R Impedance Mismatch**
- **Inheritance**
- **Queries**

O/R Impedance Mismatch

	DB	OO
• **Identity**	PK	`a == b` `a.equals(b)`
• **Associations**	FK	references, fields bidirectional redundancy
• **Inheritance**	???	`extends`
• **Navigation**	JOINs	`a.b.c` field access



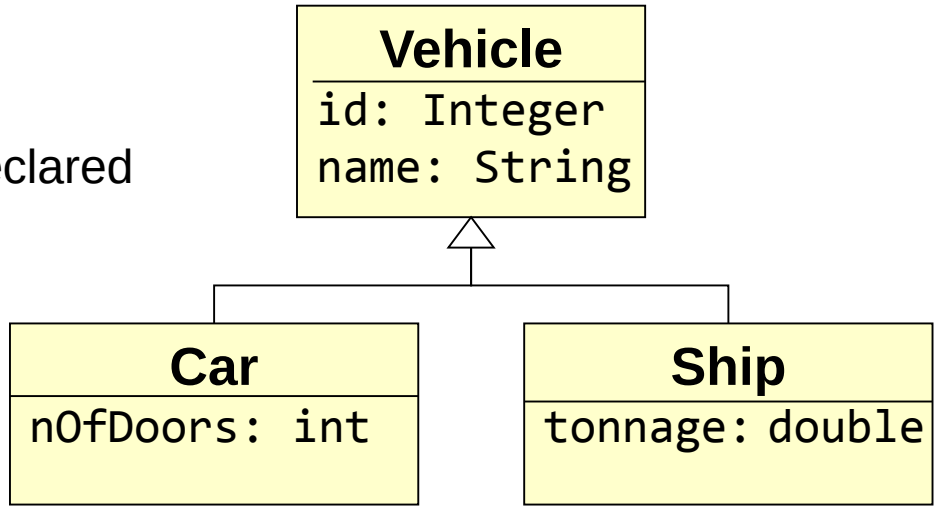


Java Persistence API

- **O/R Impedance Mismatch**
- **Inheritance**
- **Queries**

Inheritance

- **Entities**
  - All classes have to be declared as `@Entity` classes



- **Representation**
  - Inheritance type can be specified on root entity using `@Inheritance` annotation
    - `SINGLE_TABLE` default
    - `TABLE_PER_CLASS` per concrete class a table is defined
    - `JOINED` one table per class

Inheritance

- **@Inheritance**

```
public @interface Inheritance {
  /** The strategy to be used for the entity inheritance hierarchy.*/
  InheritanceType strategy() default SINGLE_TABLE;
}
```

```
public enum InheritanceType {
  SINGLE_TABLE,    // A single table per class hierarchy
  TABLE_PER_CLASS, // A table per concrete entity class
  JOINED // A strategy in which fields that are specific to a
         // subclass are mapped to a separate table than the fields
         // that are common to the parent class, and a join is
         // performed to instantiate the subclass.
}
```

Inheritance

- **SINGLE_TABLE**
  - `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`

DTYPE	ID	NAME	NOFDOORS	TONNAGE
Car	1	VW Sharan	5	(null)
Car	2	Smart	2	(null)
Ship	3	Queen Mary	(null)	76000

  - All attributes are stored in one table
  - Type is stored in a discriminator column
    - `@DiscriminatorColumn(String name, DiscriminatorType type)`
      - name: Name of the column (default: DTYPE)
      - type: Type of the column (STRING, CHAR, INTEGER)
  - Using the annotation @DiscriminatorValue discriminator can be specified
    - Default is the unqualified class name
  - All fields added in subclasses must be nullable
  - Foreign keys can only refer to the base class (Ship is not known in DB)

Inheritance

- **JOINED**
  - `@Inheritance(strategy = InheritanceType.JOINED)`

ID	NAME
1	VW Sharan
2	Smart
3	Queen Mary

ID	NOFDOORS
1	5
2	2

ID	TONNAGE
3	76000

  - A table is defined for *each* class, primary key is joined
  - Primary key is also a foreign key to the base table
  - Advantages:
    - Normalized schema, a database table for each class
    - All fields can be defined with *not null* conditions
    - Foreign-key references to concrete subclasses are possible
  - Disadvantages:
    - Each entity access has to go over several tables

Inheritance

- **TABLE_PER_CLASS**
  - `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`

ID	NAME	NOFDOORS
1	VW Shara	5
2	Smart	2

ID	NAME	TONNAGE
3	Queen Mary	76000

  - A table is defined for each non-abstract class which contains the attributes of this class and of all base classes
  - Advantages:
    - Non-null constraints can be defined
    - Foreign-key references to concrete subclasses are possible (but not to abstract base classes)
  - Disadvantages:
    - Polymorphic queries need to access several tables
    - Identity generator cannot be used
    - Support is optional for JPA 2.2 (but provided by Hibernate)

Inheritance

- **Non-Entity base classes**
  - Entity classes may be derived from non-domain classes which should not be mapped to database tables => `@MappedSuperClass`

```
@MappedSuperclass
public abstract class UuidEntity {
  @Id protected String id;

  public UuidEntity() { this.id = UUID.randomUUID().toString(); }

  public String getId() { return id; }
  public boolean equals(Object x) {
    return x instanceof UuidEntity
           && ((UuidEntity)x).id.equals(id);
  }
  public int hashCode() { return id.hashCode(); }
}
```

Inheritance Remarks

- **Transient**
  - If an entity class extends a class which is neither marked as `@Entity` nor as `@MappedSuperclass`
    - Those attributes are NOT persisted
    - Similar as if the fields were declared `@Transient` or `transient`
- **Convention over Configuration**
  - If no inheritance-specific annotations are made, then the default is
    - Inheritance Strategy: SINGLE_TABLE
    - Discriminator-Type: String
    - Discriminator-Column: DTYPE
    - Discriminator-Value: unqualified class name
- **Performance**
  - In deep class hierarchies JOINED may lead to unacceptable performance

Java Persistence API

- **O/R Impedance Mismatch**
- **Inheritance**
- **Queries**
  - JPQL: Java Persistence Query Language
  - JPQL: Joins
  - Criteria API

Java Persistence Query Language

- **JPQL**
  - JPQL is used to define searches and bulk updates and deletes of persistent entities *independent of the underlying database!*
  - JPQL is inspired by SQL, but it operates directly on the entities and its fields, as defined in the Java code, rather than with database tables.
  - The main difference to SQL is that JPQL queries result in an entity (or a collection of entities) rather than in a table.

  - Characteristics
    - Dot-Notation (`rental.movie.priceCategory`) can be used instead of JOINs (where possible)
    - Entity names (not table names) are used in the queries (i.e. the name defined in an `@Entity` annotation)

JPQL: Statements

- **Select Statement**

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

SELECT defines the format of the query results

FROM defines the entity (or entities) from which the results will be obtained

The remaining clauses can be used to restrict or order the result of a query

- **Bulk Update**

```
UPDATE <entity name> [[AS] <identification variable>]
SET <update statement> {, <update statement>}
[WHERE <where clause>]
```

- **Bulk Delete**

```
DELETE FROM <entity name> [[AS] <identification variable>]
[WHERE <where clause>]
```

JPQL: Select and From Clauses

- **Select Statement**

```
SELECT c FROM Customer c
```

```
SELECT c.name FROM Customer c
```

```
SELECT c FROM Customer c WHERE c.address.city = 'Basel'
```

```
SELECT c.address FROM Customer c
```

```
SELECT c.name, c.prename FROM Customer c
```

  - Result is of type Object[] (of length 2) or List<Object[]>

```
SELECT DISTINCT c.address.city FROM Customer c
```

  - DISTINCT removes duplicates

```
SELECT NEW ch.fhnw.edu.PersonDto(c.name, c.prename) FROM Customer c
```

  - Allows to create arbitrary objects (also non-Entity classes)

```
SELECT pk FROM PriceCategory pk
```

  - Polymorphic select statements are possible

JPQL: Where Clause

- **The WHERE clause of a query is used to specify filtering conditions to reduce the result set**
  - `=, <=, >=, <, >, <>`
    ```
    WHERE c.name = 'Meier'
    ```
  - `[NOT] BETWEEN … AND …`
    ```
    WHERE c.age between 20 and 30
    ```
    for numeric, strings and date expressions
  - `[NOT] LIKE …`
    ```
    WHERE c.name LIKE 'A%'
    ```
    (% and _)

    pattern matches are case sensitive!
  - `[NOT] IN (…)`
    ```
    WHERE c.phonePrefix IN ('079','078')
    ```
  - `IS [NOT] NULL`
    ```
    WHERE c.adr IS NOT NULL
    ```
  - `IS [NOT] EMPTY`
    ```
    WHERE c.rentals IS NOT EMPTY
    ```
  - `[NOT] MEMBER OF`
    ```
    WHERE 'JPA' MEMBER OF c.skills
    ```
    ```
    WHERE :project MEMBER OF e.projects
    ```
    ```
    WHERE e MEMBER OF e.knows
    ```
  - `NOT, AND, OR`

JPQL: Subqueries

- **Subqueries**
  - Subqueries can be used in the WHERE (or HAVING) clause of a query

```
SELECT e FROM Employee e
   WHERE e.salary = (SELECT MAX(emp.salary) FROM Employee emp)
```

  - [NOT] EXISTS (<subquery>)
    - *tests whether a subquery returns a result*

```
SELECT e FROM Employee e WHERE EXISTS
   (SELECT 1 FROM Phone p
    WHERE p.employee = e AND p.type = 'Cell')
```

  - [NOT] IN (<subquery>)

```
SELECT e FROM Employee e WHERE e.department IN
   (SELECT DISTINCT d FROM Department d
         JOIN d.employees de JOIN de.projects p
         WHERE p.name LIKE 'QA%')
```

JPQL: Functions

- **Functions can be used in SELECT and WHERE clauses**
  - Strings:
    - CONCAT, SUBSTRING, TRIM, LOWER, UPPER, LENGTH, LOCATE
  - Math functions:
    - ABS, SQRT, MOD
  - Many Associations:
    - SIZE, INDEX
  - Temporal:
    - CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP()
  - Conditional:
    -
```
CASE
    WHEN … THEN …
    WHEN … THEN …
    ELSE …
END
```

JPQL: Aggregate Functions

- **Aggregate Functions can be defined in the select clause**
  - AVG => java.lang.Double
  - COUNT => java.lang.Long
  - MAX => type of the field on which max is applied
  - MIN => type of the field on which min is applied
  - SUM => Double, Long, BigDecimal, BigInteger

```
SELECT MAX(c.age) FROM Customer c
```

```
SELECT COUNT(r) FROM Rental r WHERE r.user.name = :name
```

```
SELECT u FROM User u WHERE u.birthdate = (
                           SELECT MIN(u2.birthdate) FROM User u2)
```

JPQL: Select Statment

- **Syntax according to the specification**

```
select_clause ::= SELECT [DISTINCT] select_item {, select_item}*
select_item ::= select_expression [[AS] result_variable]
select_expression ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable |
    OBJECT(identification_variable) |
    constructor_expression
constructor_expression ::=
    NEW constructor_name (constructor_item {, constructor_item}*)
constructor_item ::=
    single_valued_path_expression |
    scalar_expression |
    aggregate_expression |
    identification_variable
aggregate_expression ::=
    {AVG | MAX | MIN | SUM} ([DISTINCT] state_valued_path_expression) |
     COUNT ([DISTINCT] identification_variable | state_valued_path_expression |
         single_valued_object_path_expression) |
     function_invocation
```

Query API: Dynamic Queries

- **Dynamic Query**
  - Dynamic queries (JPQL queries specified at runtime) are created with the method `createQuery` on an entity manager
    - `em.createQuery(String q)` => returns a un-typed query
    - `em.createQuery(String q, Class<T> c)` => returns a typed query
  - Accessing results:
    - `q.getResultList()` // static type of result is a list (un-typed or typed)
    - `q.getResultStream()` // returns `q.getResultList().stream();`
    - `q.getSingleResult()` // result of type Object or of the expected type T
      - `NoResultException` if no entry was found
      - `NonUniqueResultException` if several entities were found

```
TypedQuery<Movie> q = em.createQuery(
   "SELECT m from Movie m WHERE m.title = :title", Movie.class);
q.setParameter("title", title);
List<Movie> movies = q.getResultList();
```

Query API: Named Queries

- **Named Queries**
  - Queries may be defined on entity classes (and can be parsed ahead)

```
@NamedQuery(name="Movie.all", query="SELECT m FROM Movie m")
@NamedQuery(name="Movie.byTitle",
      query="SELECT m FROM Movie m WHERE m.title = :title")
class Movie {...}
```

    - `NamedQuery`-Annotation is declared to be repeatable since JPA2.2
      - `NamedQueries` wrapper annotation is no longer needed
    - Query names are global (i.e. scoped by the persistence unit)
      - => names must be unique; common practice is to prefix with entity name
  - Queries are created with `createNamedQuery` on an entity manager

```
TypedQuery<Movie> q = em.createNamedQuery(
                                  "Movie.byTitle", Movie.class);
q.setParameter("title", title);
List<Movie> movies = q.getResultList();
```

Query API: Named Queries

- **Named Queries**
  - Problem: Name of the queries are strings, i.e. not type safety
    - Typos are not detected by the compiler and no refactoring support
  - Convention: definition of constants

```
@NamedQuery(name = Movie.FIND_ALL, query="SELECT m FROM Movie m")
@NamedQuery(name = Movie.FIND_BY_TITLE,
            query = "SELECT m FROM Movie m WHERE m.title=:title")
public class Movie {
   public static final String FIND_ALL = "Movie.all";
   public static final String FIND_BY_TITLE = "Movie.byTitle";
   …
```

```
public List<Movie> findByTitle(String title) {
   return em.createNamedQuery(Movie.FIND_BY_TITLE, Movie.class)
      .setParameter("title", title)
      .getResultList();
}
```

Query API: Parameters

- **Positional Parameters**

```
"SELECT m FROM Movie m WHERE m.title = ?1"
```

  - Actual value is set with `q.setParameter(int, Object)`
    - Returns the query => fluent interface
  - Numbering starts with 1

```
q.setParameter(1, "No Time To Die");
```

- **Named Parameters**

```
" SELECT m FROM Movie m WHERE m.title = :title"
```

  - Actual value is set with `q.setParameter(String, Object)`
    - Returns the query => fluent interface

```
q.setParameter("title", "No Time To Die");
```

JPQL: Ordering and Paging

- **Order-By clause allows to order the result**
  - One or several sorting fields
  - ASC or DESC (ASC = default)

```
SELECT c FROM Customer c WHERE c.age > 18
ORDER BY c.age DESC, c.address.country.code ASC
```

- **Query returning a list result can be restricted to a range**
  - `public Query setMaxResults(int maxResult)`
  - `public Query setFirstResult(int startPosition)`
    - Start with 0

```
TypedQuery<Movie> q = em.createQuery(
          "SELECT m FROM Movie m ORDER BY m.name", Movie.class)
    .setFirstResult(20)
    .setMaxResults(10);
List<Movie> movies = q.getResultList();
```

Ordering and Paging: Generated SQL

```
TypedQuery<Movie> q = em.createQuery(
         "SELECT m FROM Movie m ORDER BY m.name", Movie.class)
   .setFirstResult(20)
   .setMaxResults(10);
List<Movie> movies = q.getResultList();
```

- **H2, Postgres (with Hibernate)**

```
SELECT * FROM Movie movie0_
   ORDER BY movie0_.name
   LIMIT 10
   OFFSET 20
```

- **MySQL (with Hibernate)**

```
SELECT * from Movie movie0_
   ORDER BY movie0_.name
   LIMIT 20, 10
```

Ordering and Paging: Generated SQL

```
TypedQuery<Movie> q = em.createQuery(
         "SELECT m FROM Movie m ORDER BY m.name", Movie.class)
   .setFirstResult(20)
   .setMaxResults(10);
List<Movie> movies = q.getResultList();
```

- **Oracle (with Hibernate)**

```
SELECT * FROM
  (
    SELECT row_.*, rownum rownum_ from
      (
        SELECT * FROM Movie movie0_ ORDER BY movie0_.name
      ) row_
    WHERE rownum <= 30
  )
WHERE rownum_ > 20
```

Update- and Delete-Statements

- **Bulk Updates / Bulk Deletes**
  - With JPA update and delete operations can be performed without creating instances
  - Can be applied on one entity only (no joins)
  - Entities which are loaded in a entity context are not affected
    - => bulk updates should be executed in a separate transaction
    - Result returns the number of changed or deleted entries

```
Query q = em.createQuery(
   "DELETE FROM Movie m WHERE m.id > 1000");
int result = q.executeUpdate();
```

```
Query q = em.createQuery(
   "UPDATE User u SET u.name = 'TOO YOUNG' WHERE c.age < 18");
int result = q.executeUpdate();
```

Java Persistence API

- **O/R Impedance Mismatch**
- **Inheritance**
- **Queries**
  - JPQL: Java Persistence Query Language
  - JPQL: Joins
  - Criteria API

Implicit Joins

- **ManyToOne / OneToOne**

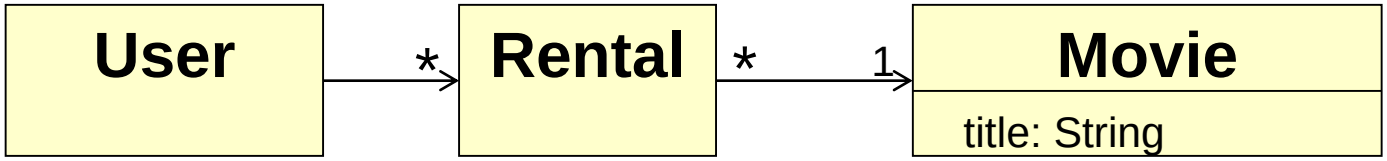
```
SELECT c.name, c.address.city FROM Customer c
```

  - Join is performed implicitly by the data base
  - Implicit joins are always inner joins

```
Hibernate:
   select c.name as col0, a.city as col1
   from Customer c, Address a
   where c.address_id=a.id
```

Inner Joins

- **Associations across XxxToMany-Associations**



  - **SELECT u.rentals.movie.title FROM User u** does not work

```
SELECT r.movie.title from User u JOIN u.rentals r
SELECT r.movie.title from User u INNER JOIN u.rentals r
SELECT r.movie.title from User u, IN(u.rentals) r
```

  - Performs an inner join => entries may be duplicated (if different users refer to the same movie)
    - Duplicates can be removed with SELECT DISTINCT
  - Inner join only returns entities which are referenced

Outer Joins

- **Inner vs Outer Joins**
  - Inner join only returns entities which are part of an association, whereas outer join returns objects which have no references / are not referenced
  - Outer Joins: JPA support only left outer joins
  - Example
    - Inner Join

```
SELECT u.name, r FROM User u JOIN u.rentals r
```

      - Returns name and rentals only from those users which have rented movies
      - Syntax: `[INNER] JOIN` *`path-expression variable`*
    - Outer Join

```
SELECT u.name, r FROM User u LEFT JOIN u.rentals r
```

      - Returns all users (r may then be null)
      - Syntax: `LEFT [OUTER] JOIN` *`path-expression variable`*

Fetch Joins

- **Fetch Joins and Lazy loading**
  - Allows to eagerly load dependent objects, i.e. allows to redefine the loading strategy *for a query*

```
SELECT u from User u
```

    - Rentals are not loaded (if not defined as eager loading)

```
SELECT u from User u LEFT JOIN FETCH u.rentals
```

    - Query which loads the rentals and which returns ALL users (also those which do not have rentals) as it is an outer join
    - Inner join would be possible as well (but less useful)
  - Syntax:
    - `[ LEFT [OUTER] | INNER] JOIN FETCH` *`path-expression`*

Java Persistence API

- **O/R Impedance Mismatch**
- **Inheritance**
- **Queries**
  - JPQL: Java Persistence Query Language
  - JPQL: Joins
  - Criteria API

Criteria API

- **Problems**
  - Typing errors only reveal themselves at runtime
    - `SELECT m from Movie`

```
java.lang.IllegalStateException: No data type for node:
org.hibernate.hql.internal.ast.tree.IdentNode
 \-[IDENT] IdentNode: 'm' {originalText=m}
```

    - `SELECT u.firstname from User u`

```
org.hibernate.QueryException: could not resolve property:
firstname of: ch.fhnw.eaf.jpa.model.User [SELECT u.firstname
from ch.fhnw.eaf.jpa.model.User u]
```

- **Solution: Criteria API**
  - Object-oriented builder for queries
  - Provides a fluent interface

Criteria API

- **Example: find movie by title**

```
public List<Movie> getByTitle(String title) {
   CriteriaBuilder builder = em.getCriteriaBuilder();
   CriteriaQuery<Movie> query = builder.createQuery(Movie.class);

   // SELECT m FROM Movie m
   Root<Movie> m = query.from(Movie.class);
   query.select(m);

   // WHERE m.title = :title
   Path<String> p = m.get("title");
   Predicate pred = builder.equal(p, title);
   query.where(pred);

   return em.createQuery(query).getResultList();
}
```

Criteria API

- **Example: find movie by title**

```
public List<Movie> getByTitle(String title) {
   CriteriaBuilder builder = em.getCriteriaBuilder();
   CriteriaQuery<Movie> query = builder.createQuery(Movie.class);

   // SELECT m FROM Movie m WHERE m.title = :title
   Root<Movie> m = query.from(Movie.class);
   query.select(m).where(builder.equal(m.get("title"), title));

   return em.createQuery(query).getResultList();
}
```

  - SELECT, FROM and WHERE have an API representation
  - CriteriaBuilder contains methods to construct the query
    - Factory for the criteria query (with a particular result type)
    - asc, desc, avg, sum, max, min, count, and, or, greaterThan, lowerThan, …
  - Problem with the string remains (`m.get("title")`)

Criteria API: Static Meta-model

- **Meta data class E_ for each entity class E**

```
@StaticMetamodel(Movie.class)
public abstract class Movie_ {
  public static volatile SingularAttribute<Movie, Boolean> rented;
  public static volatile SingularAttribute<Movie, LocalDate>
                                                releaseDate;
  public static volatile SingularAttribute<Movie, PriceCategory>
                                                priceCategory;
  public static volatile SingularAttribute<Movie, Long> id;
  public static volatile SingularAttribute<Movie, String> title;
}
```

  - Name is a naming convention (connection is done over the annotation)
  - Meta-model contains name and type of the fields
  - `ListAttribute` represents `OneToMany/ManyToMany` associations

```
  public static volatile ListAttribute<User, Rental> rentals;
```

Criteria API: Static Meta-model

- **Example: find movie by title**

```
public List<Movie> getByTitle(String title) {
   CriteriaBuilder builder = em.getCriteriaBuilder();
   CriteriaQuery<Movie> query = builder.createQuery(Movie.class);

   // SELECT m FROM Movie m WHERE m.title = :title
   Root<Movie> m = query.from(Movie.class);
   query.select(m).where(builder.equal(m.get(Movie_.title), title));

   return em.createQuery(query).getResultList();
}
```

  - After a refactoring of class `Movie` errors can be detected

Criteria API: Static Meta-model

- **Example: INNER JOIN**
  - SELECT u.lastName, r FROM User u JOIN u.rentals r

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Object[]> query =
                        builder.createQuery(Object[].class);

Root<User> user = query.from(User.class);
Join<User, Rental> rental = user.join(User_.rentals);
query.select(builder.array(
   user.get(User_.lastName),
   rental.get(Rental_.movie).get(Movie_.title)
));

List<Object[]> result = em.createQuery(query).getResultList();

for (Object[] res : result) { … }
```

Returns a list of arrays of length two of all the lastName-title associations

Criteria API: Static Meta-model: Gradle

```
dependencies {
 …
 annotationProcessor('org.hibernate:hibernate-jpamodelgen:5.5.7.Final')
}

tasks.withType(JavaCompile) {
  options.annotationProcessorGeneratedSourcesDirectory =
                                    file("$buildDir/generated/")
}

sourceSets {
  main {
    java {
      srcDirs += ["$buildDir/generated/"]
    }
  }
}
```

Java Persistence API

- **Inheritance**
- **Queries**
  - JPQL: Java Persistence Query Language
  - JPQL: Joins
  - Criteria API