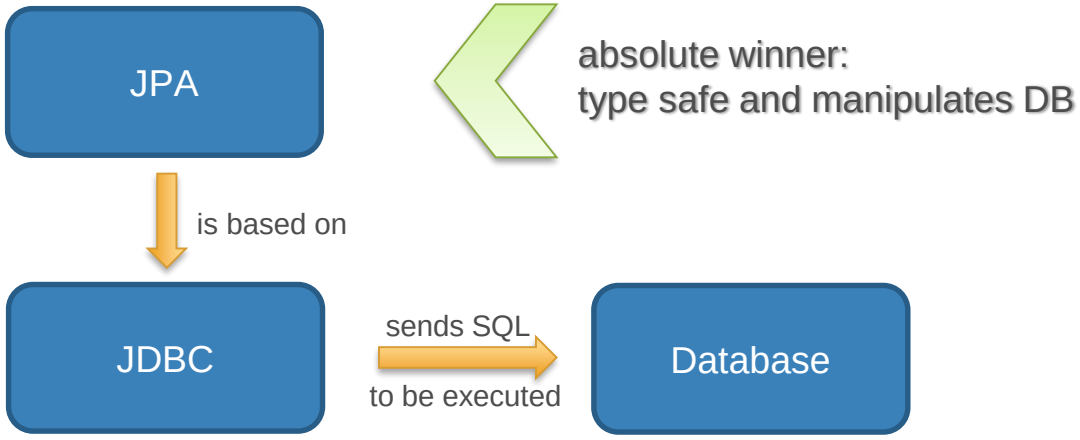
Java Persistence API – part 2

Entity Annotations, Relationships (part 2)

Tamás Ambrus

Revision: JPA part 1

JPA mostly relies on JDBC, it is a standard for ORM, so this technology is able to map between objects in code and database table records.



More info at https://stackoverflow.com/questions/11881548/jpa-or-jdbc-how-are-they-different

Entity Annotations

Table, Column, Enumerated, Primary keys

@Table, @Column

We can use these annotations to override default table and column properties.

```java
@Entity
@Table(name = "employee_table")
public class Employee {

    @Id
    private int id;

    @Column(nullable = false, length = 50, updatable = false)
    private String name;

    public Employee() {
    }

    public Employee(String name) {
        this.name = name;
    }
```

More info at https://docs.oracle.com/javaee/7/api/javax/persistence/Table.html
https://docs.oracle.com/javaee/7/api/javax/persistence/Column.html

@Basic

This annotation knows nothing more than setting fetch and optionality.

```
@Entity
@Table(name = "employee_table")
public class Employee {

    @Id
    private int id;

    @Basic(optional = false, fetch = FetchType.LAZY)
    private String name;

    public Employee() {
    }

    public Employee(String name) {
        this.name = name;
    }
```

`Eager` fetching loads the value as soon as it's possible
`Lazy` fetching loads only if needed (upon code)

Note that `LAZY` is only a hint, JPA won't necessarily fetch lazily.

Note that attributes that are in collection-valued relationship, tend to be fetched lazily. Other attributes fetch eagerly.

More info at https://docs.oracle.com/javaee/7/api/javax/persistence/Basic.html

@Enumerated

We use this annotation for manipulating an enumeration typed object.

```java
public enum EmployeeType { FULL, PART, CONTRACT }

@Entity
@Table(name = "employee_table")
public class Employee {

    @Id
    private int id;

    @Enumerated(EnumType.STRING)
    private EmployeeType type;

    public Employee() {
    }
```

Enumerations can be stored in 2 ways: by their name and by their index (STRING, ORDINAL)

Which one is better?

TYPE
FULL
CONTRACT
NULL

OR

TYPE
0
2
NULL

More info at https://docs.oracle.com/javaee/7/api/javax/persistence/Enumerated.html

Primary keys

Entities must have primary keys: either single or composite ones. We mostly use generated single primary keys.

Primary keys must have one of these types:

- primitive types, wrapper types, String
- java.util.Date, java.sql.Date
- java.math.BigDecimal, java.math.BigInteger

It is always easy to generate a primary key with JPA instead of hardly find one of our class fields. There are 4 ways to generate primary keys.

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro001.htm#JEETT01152

Primary keys: single, GenerationType.AUTO

This method should only be used during development. It is suggested to change this to a concrete strategy before releasing the product.

```java
@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String name;

    public Employee() {
    }
```

This option lets the persistence provider to pick an appropriate strategy for the database.
(so 1 of the following strats)

More info at https://docs.oracle.com/javaee/7/api/javax/persistence/GenerationType.html

Primary keys: single, GenerationType.IDENTITY

The id column with this strategy will be set automatically to the larger-by-one value. Note that the id field has no valid value until it's inserted into the database.

```
@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    public Employee() {
    }
```

More info at https://docs.oracle.com/javaee/7/api/javax/persistence/GenerationType.html

Primary keys: single, GenerationType.SEQUENCE

This strategy creates a sequence in the database which will provide unique values. Note that the id field has no valid value until it's inserted into the database.

```
@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private int id;

    private String name;

    public Employee() {
    }
```

More info at https://docs.oracle.com/javaee/7/api/javax/persistence/GenerationType.html

Primary keys: single, GenerationType.TABLE

This strategy is customizable in depth. It creates a different table to help generate the keys.

```java
@Entity
public class Employee {

    @Id
    @TableGenerator(name = "ID_Generator", table = "ID_GEN_TABLE",
        pkColumnName = "ID_NAME", valueColumnName = "NEXT_ID_GEN_SOURCE",
        pkColumnValue = "Employee_ID", initialValue = 1, allocationSize = 64)
    @GeneratedValue(strategy = GenerationType.TABLE, generator = "ID_Generator")
    private int id;
```

Employee:

	ID	NAME
	1	NULL
	65	NULL
	66	NULL
	NULL	NULL

Id_gen_table:

	ID_NAME	NEXT_ID_GEN_SOURCE
	Emplovee ID	128
	NULL	NULL

More info at https://docs.oracle.com/javaee/7/api/javax/persistence/GenerationType.html

Primary keys: composite

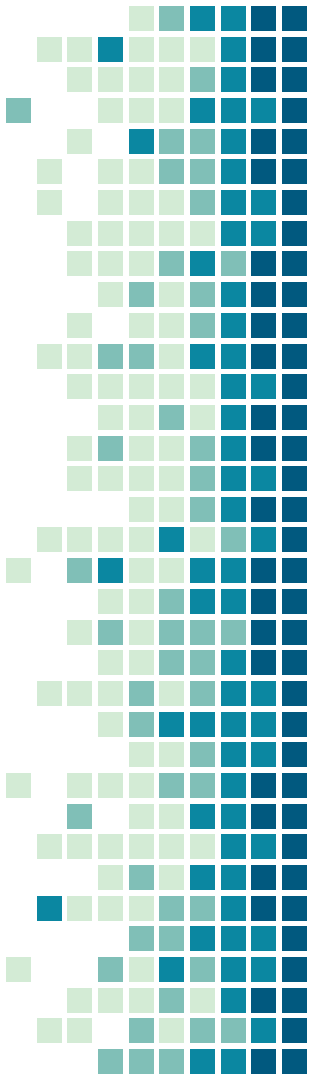
We may create composite primary keys too. These consist of multiple fields. The type – that behaves as a composite key – must meet some conditions:

- it must be public, needs a public, no-argument constructor
- it must implement the `Serializable` interface
- it must override the `equals` (and thus the `hashCode`) method

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro001.htm#JEETT01152

Primary keys: composite, example

```java
@Embeddable
public class RoomID implements Serializable {

    private String building;
    private short number;

    @Override
    public boolean equals(Object other) {
        // ...
    }

    @Override
    public int hashCode() {
        // ...
    }
}

@Entity
public class Room {

    @EmbeddedId
    private RoomID id;
    private short capacity;

    // ...
}
```



Relationships (part 2)

Cascade operations, Entity Inheritance

Cascade operations

We have learned about 4 relationships last week: one-to-one, one-to-many, many-to-one and many-to-many.

All of these support cascading. This helps us maintaining the database as some operations can be executed automatically.

For example, deleting a record from the database – that is in relation with a record of another table – may cause the other record to be deleted.

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro001.htm#JEETT00676

Cascade operations – common operations

We will need mostly these operations to use:

- ALL: all cascade operations will be propagated to the associated entity. ALL is equivalent to {DETACH, MERGE, PERSIST, REFRESH, REMOVE}
- PERSIST: if the current entity is being persisted to the persistence context, the associated one will be persisted too
- REMOVE: if the current entity is being removed from the persistence context, the associated one will be removed too

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro001.htm#JEETT00676

Cascade operations – example

```java
public enum EmployeeType { FULL, PART, CONTRACT }

@Entity
public class Employee {

    @Id
    @GeneratedValue
    private int id;

    @Enumerated
    private EmployeeType type;

    @OneToOne(cascade = CascadeType.ALL)
    private Person person;

    public Employee() {
    }
```

```java
@Entity
public class Person {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    @OneToOne(mappedBy = "person")
    private Employee employee;

    public Person() {
    }
```

```java
Employee e1 = new Employee(EmployeeType.FULL, new Person("Zsofi"));
Employee e2 = new Employee(EmployeeType.CONTRACT, new Person("Tamas"));
entityManager.persist(e1);
entityManager.persist(e2);

// years go by

entityManager.remove(e1);
```

	ID	TYPE	PERSON_ID
	3	2	4
	NULL	NULL	NULL

	ID	NAME
	4	Tamas
	NULL	NULL

Entity inheritance

The remaining slides are all about how entities may inherit from types. As you will see, there are many combinations.

It's suggested to analyze:

- whether the supertype can be queried
- how the supertype and subtype is stored in the database
- which one do you need in different situations

It's worth to try them out all, compare them and memorize the most important differences.

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#BNBQN

Entity inheritance – abstract superclass

```java
@Entity
public abstract class Person {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    public Person() {
    }
```

```java
@Entity
public class Employee extends Person {
    public enum EmployeeType {
        FULL, PART, CONTRACT
    }

    @Enumerated(EnumType.STRING)
    private EmployeeType type;

    private String name;

    public Employee() {
    }

    public Employee(EmployeeType type, String name) {
        super(name);
        this.type = type;
    }
```

A new column is appended for knowing the dynamic type Person:

	ID	DTYPE	NAME	TYPE
	2	Employee	Tamas	CONTRACT
	NULL	NULL	NULL	NULL

Entity inheritance – abstract superclass

Querying the abstract entity results in a query on the subtypes.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Person> cq = cb.createQuery(Person.class);
Root<Person> person = cq.from(Person.class);
cq.select(person);
entityManager.createQuery(cq).getResultList().forEach(System.out::println);
```

Even though the list contains `Person` objects, `toString()` method iterates through dynamic types (polymorphism)

```
Employee [type=CONTRACT, person=Person [id=2, name=Tamas]]
```

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT01157

Entity inheritance – mapped superclass

Mapped superclasses are not entities, however, they can store persistent fields. Entities that inherit from a mapped superclass, inherit the persistent fields.

```
@MappedSuperclass
public class Person {

    @Id
    @GeneratedValue
    private int id;

    private String name;

    public Person() {
    }
```

Mapped superclasses:

- can't be queried
- can't be targets of entity relationships
- can't be used in EntityManager or Query operations

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT01158

Entity inheritance – non-entity superclass

Non-entity classes may be superclasses of entities.These classes may be either abstract or concrete. The fields inherited by entities are non-persistent.

Non-entity superclasses:

- can't be queried
- the relationship annotations are ignored within it
- can't be used in `EntityManager` or `Query` operations

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT01159

Entity inheritance – entity superclass

When we want to inherit from an entity, we have 3 strategies:

- both subtype and supertype fields will be managed in a single table
- for each type we manage separate tables (with redundant columns)
- for each type we manage separate tables (that need to be joined)

Either way, we need to annotate the supertype about which strat we chose.

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Person {

    @Id
    @GeneratedValue
    private int id;

    private String name;
```

```java
@Entity
public class Employee extends Person {
    public enum EmployeeType {
        FULL, PART, CONTRACT
    }

    @Enumerated(EnumType.STRING)
    private EmployeeType type;
```

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT01160

Entity inheritance – entity superclass

With InheritanceType.SINGLE_TABLE strategy we store both the supertype and subtype fields in the very same table. This table needs to have a discriminator column (that is customizable pretty well).

- all different fields will be stored in the same table (NULL values?)
- the query is fast (no join)
- this is the default inheritance mode

Person:

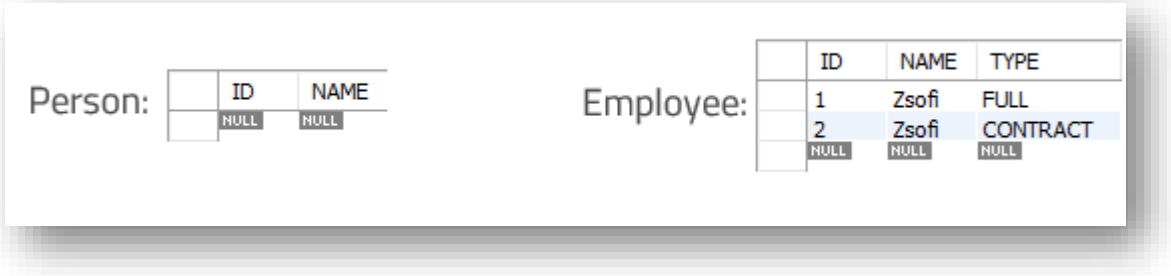
ID	DTYPE	NAME	TYPE
1	Employee	Zsofi	FULL
2	Employee	Zsofi	CONTRACT
NULL	NULL	NULL	NULL

More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT00679

Entity inheritance – entity superclass

With InheritanceType.TABLE_PER_CLASS strategy we manage separate tables for each concrete entity.

- this strategy provides poor polymorphic relationships
- the query is not fast (either UNION is needed or queries on subtypes)
- column names are inherited (we get rid of NULL values)

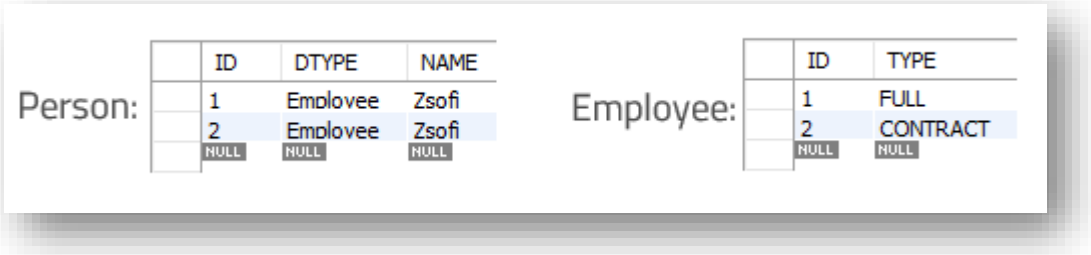


More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT00680

Entity inheritance – entity superclass

With InheritanceType.JOINED strategy we manage separate tables for each entity.

- this strategy provides very good polymorphic relationships
- the query is slow (at least one JOIN is needed – but usually more)
- fields are in their corresponding table



More info at https://docs.oracle.com/javaee/7/tutorial/persistence-intro002.htm#JEETT00681