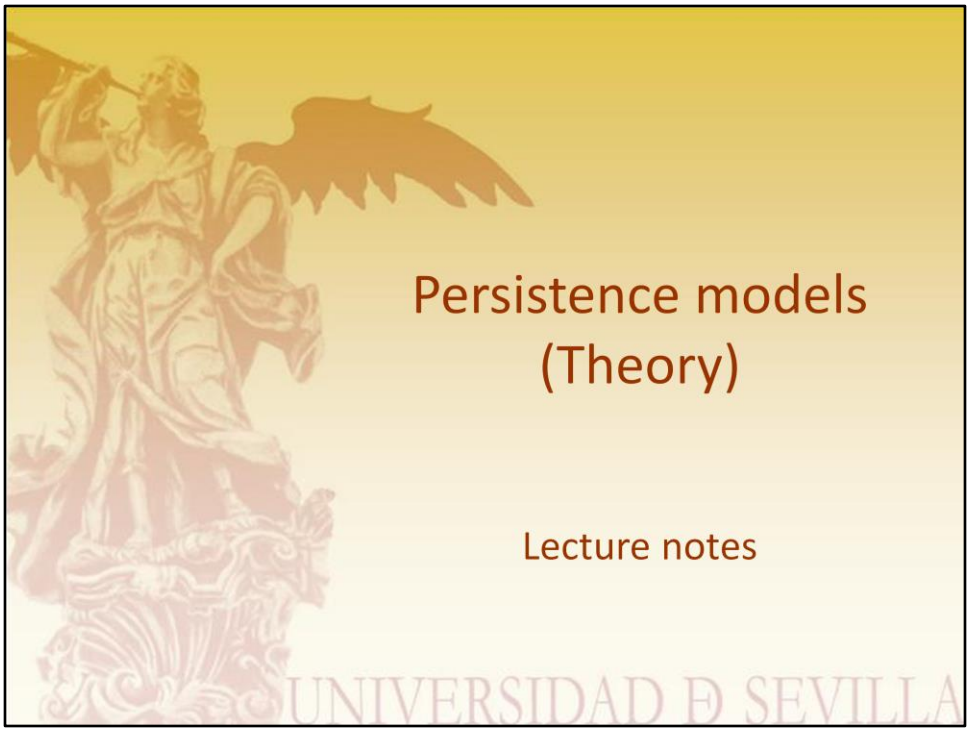



Welcome to this lecture! Today, our goal's to present the theory you need to know about persistence models.

--

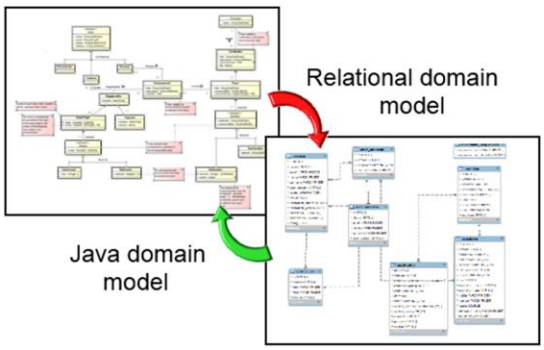
## What's a persistence model?

UNIVERSIDAD Ð SEVILLA

2

As usual, it's a good idea to start the lecture asking yourself: what's a persistence model? Please, use your intuition to produce a definition.

# This is a good definition




It's a mapping that states how Java objects must be stored and retrieved from a relational database



This is our definition: it's a mapping that states how Java objects must be stored to or retrieved from a relational database. In other words: it states how your domain objects are persisted. The emphasis is on the fact that Java objects live in memory, that is, when the server shutdowns or there are no references to them, they are disposed by the JVM garbage collector; that is the data they store is lost forever. On the contrary, the data that is stored in a database is persistent and may remain there for ages.

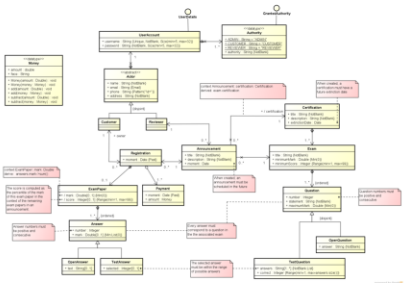
## How are they devised?

UNIVERSIDAD Ð SEVILLA

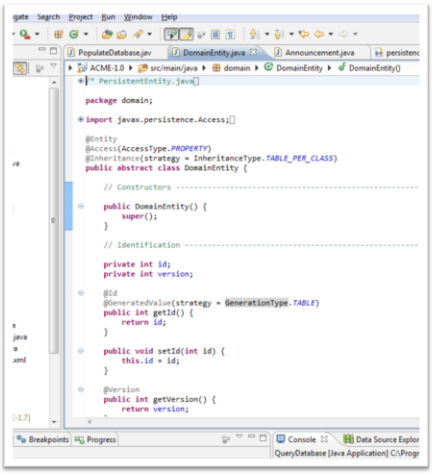
4

Before peeking at the following slides, it'd be interesting that you spent a little time at thinking how a persistence model is devised.

# Starting point: your domain model



UML domain model




Java domain model

UNIVERSIDAD D SEVILLA

5

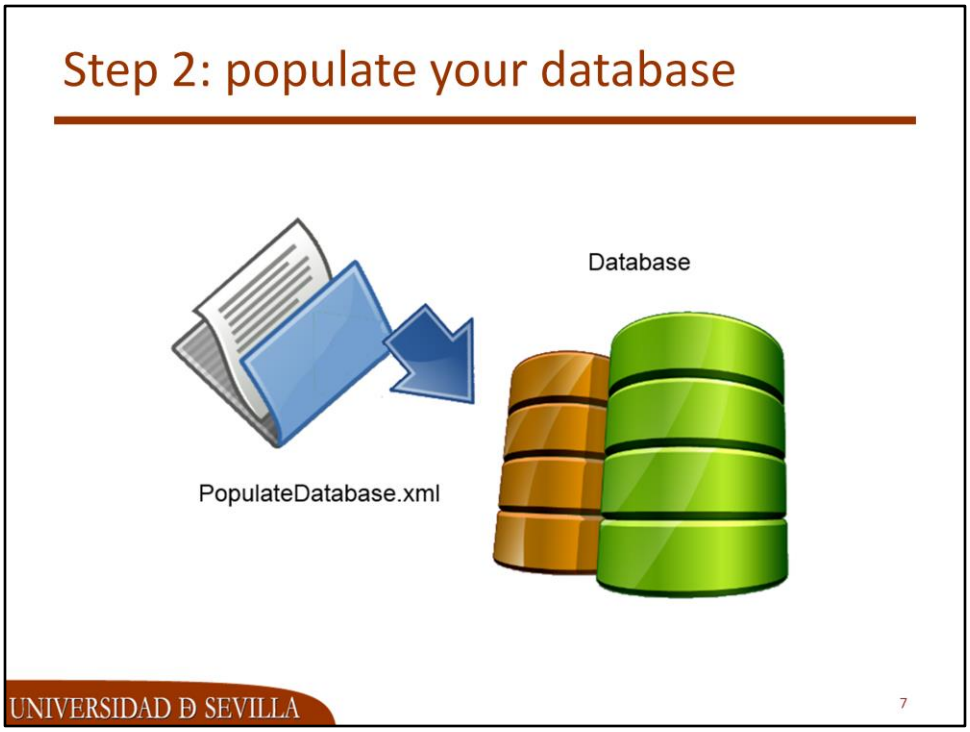
What was your conclusion? We hope it doesn't deviate too much from ours. The starting point to devise a persistence model's your UML domain model and your Java domain model: both of them. The former provides quite a handy overall view of the domain model, and the latter shall be enriched to support the persistence model.

## Step 1: create mappings

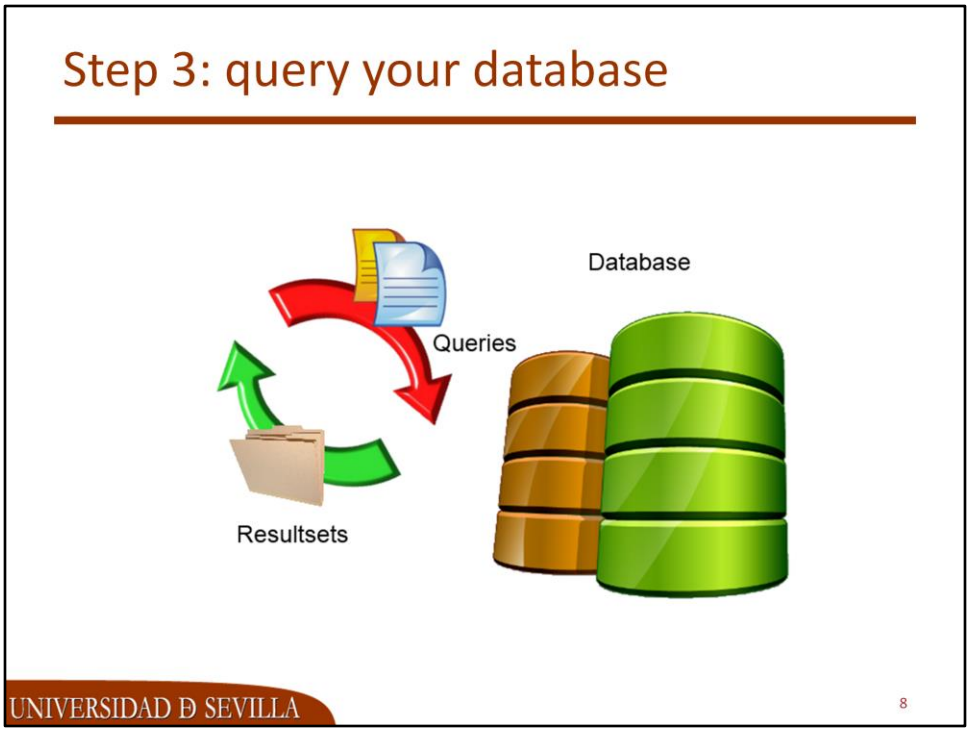

UNIVERSIDAD Ð SEVILLA

6

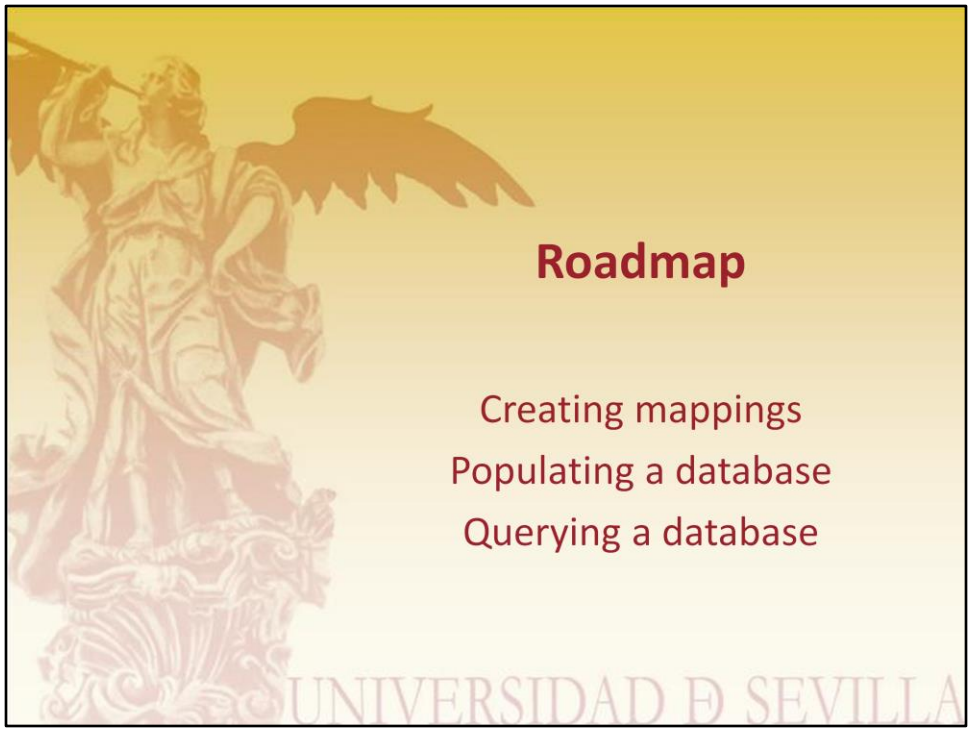
The first step is to create mappings that specify how Java objects must be mapped onto a relational database and vice versa.



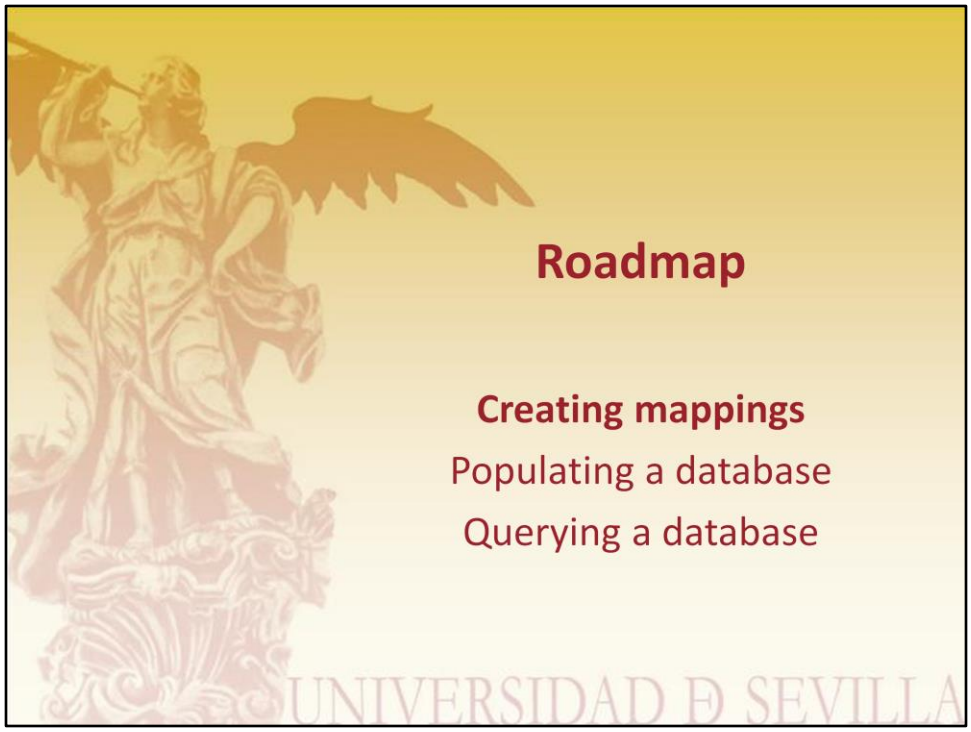

The next step is to populate your database using test data, that is sample data you'll use to test that your system works as your customer expects.




Once your database is populated with test data, it obviously makes sense that you query it, just to make sure that the population was OK.




So, it shouldn't be surprising that this is the roadmap of today's lecture: first, we'll report on how to create mappings using JPA annotations, then on how to populate a database with testing data, and, finally, we'll delve into the details regarding querying a database.



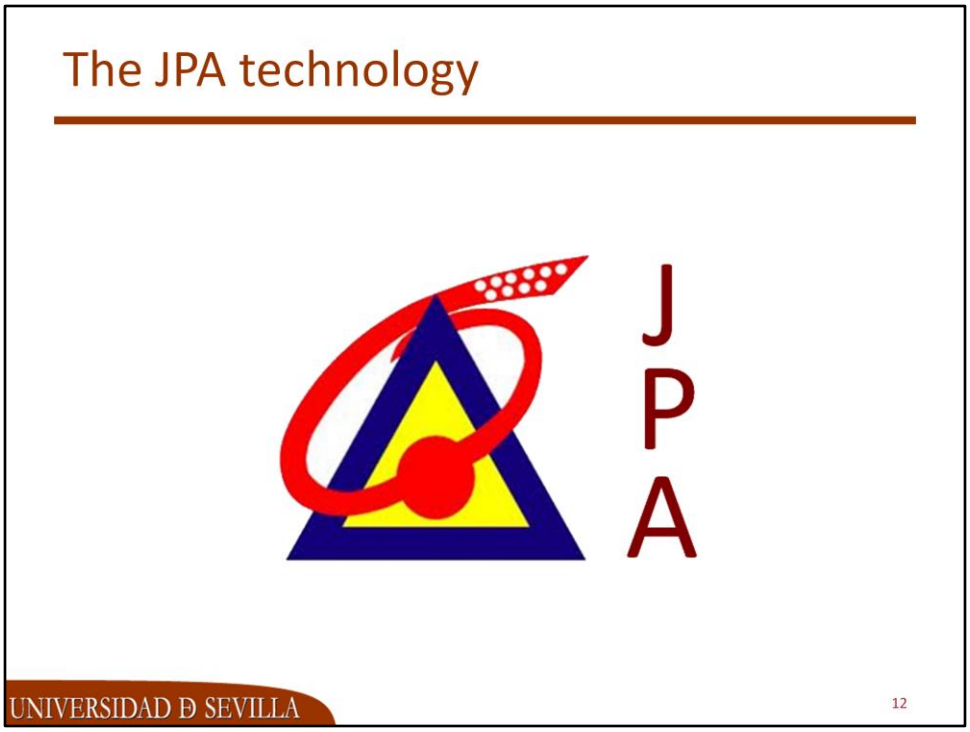

Let's start with the section on creating mappings.

## What are they?

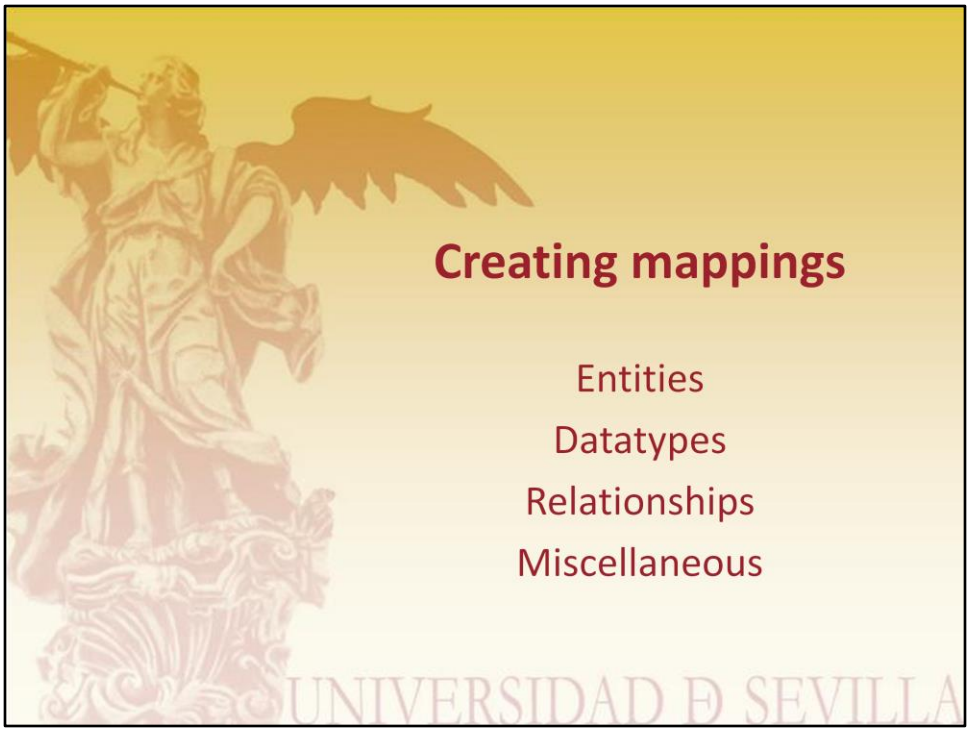

They specify how Java objects are persisted to databases and vice versa



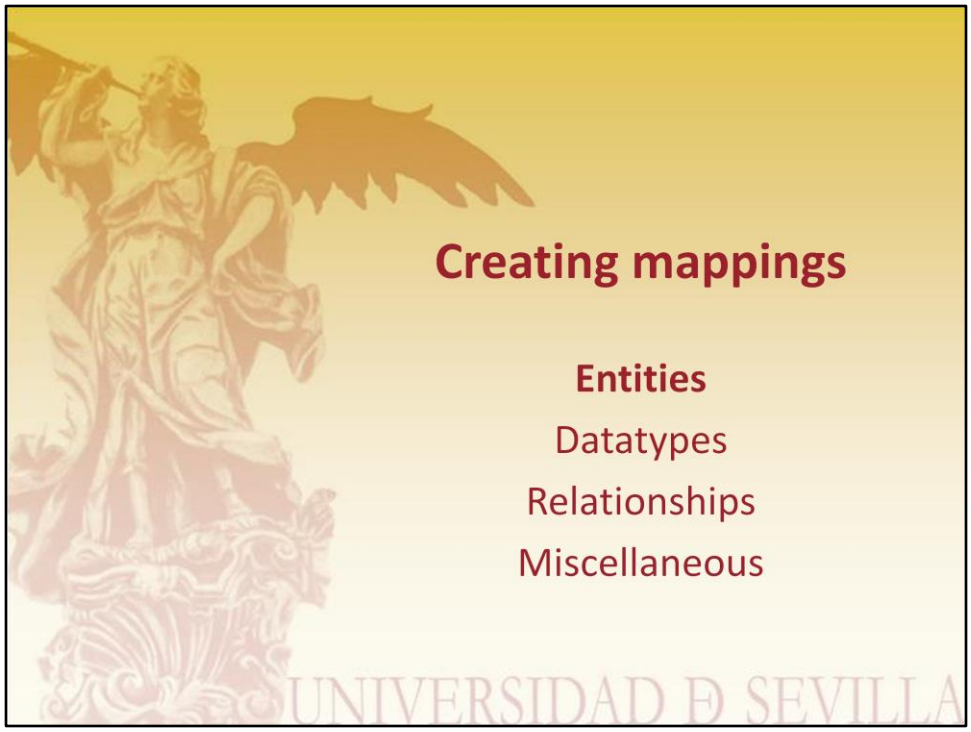
So far we know that a mapping is a specification of how Java objects are persisted to a database and vice versa, but we don't know about the technology to create them.



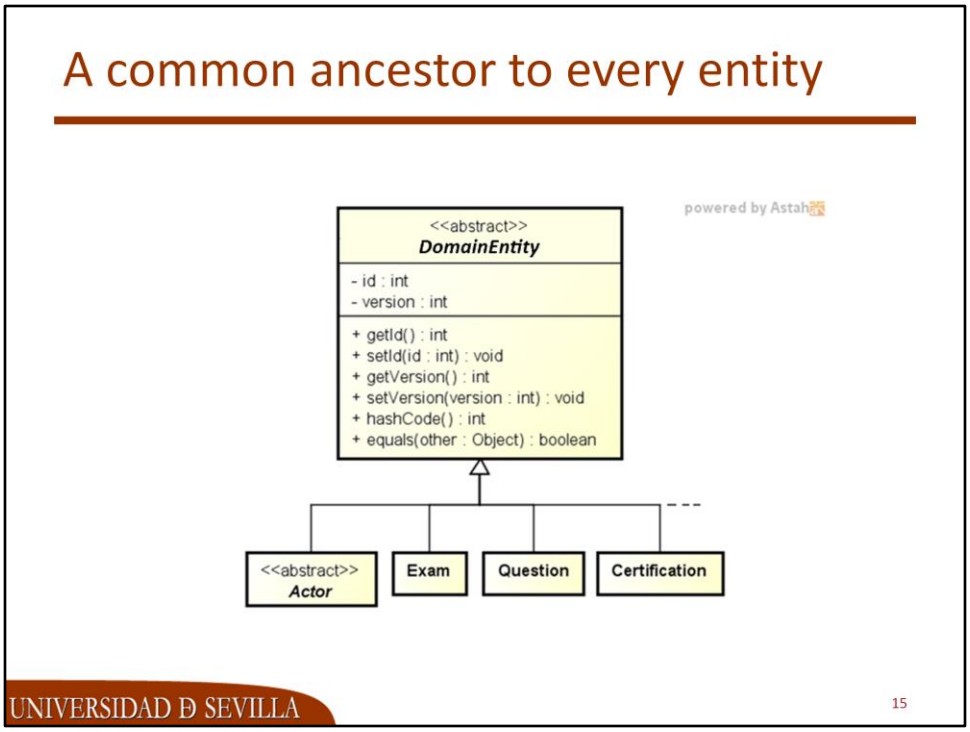

We are going to use a technology called JPA, which stands for Java Persistence Architecture. It provides a number of annotations with which we'll enrich our Java domain models so that they have explicit information about the mappings.



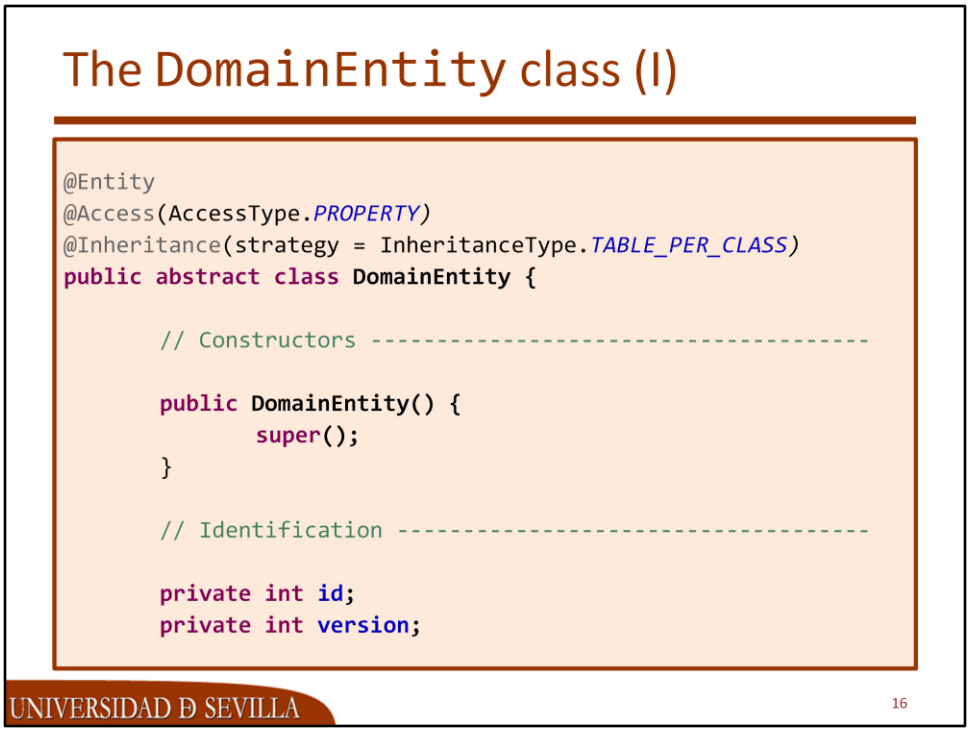

These are the sections of our roadmap regarding mappings. We'll explore how to create mappings for entities, datatypes, relationships, and a few miscellaneous mappings.




Let's start with some ideas regarding how to map entities.




From the previous lesson, you must know that our project template provides a class called “DomainEntity”, which is intended to be the common ancestor of every entity in your Java domain model. You must also know that this class endows every entity with an “id” attribute and a “version” attribute; the former represents its identification and the latter its version, that is, the number of times it’s been modified since it was created; you must also know that this class redefines the standard “equals” and “hashCode” methods. It’s now time to delve into the details. Please, keep reading.




This is an excerpt of class "DomainEntity". The key features are the following:

- We use annotation "@Entity" to indicate that this class represents an entity and that its objects can be persisted to a database. Note that "DomainEntity" is an abstract class, which means that its instances must belong to other classes that are below it in the inheritance hierarchy. For instance, objects of classes "Customer", "Exam", or "Question".
- We use annotation "@Access(AccessType.PROPERTY)" to indicate that JPA must have access to the attributes of our entities by means of the corresponding getters and setters. (By default, JPA has direct access to the attributes themselves, which isn't very appealing.)
- We also use annotation "@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)" to specify that every class that inherits from "DomainEntity" must be mapped onto its own table in the database. There are other strategies (please, take a look at the recommended bibliography), but this seems to work quite well in practice.

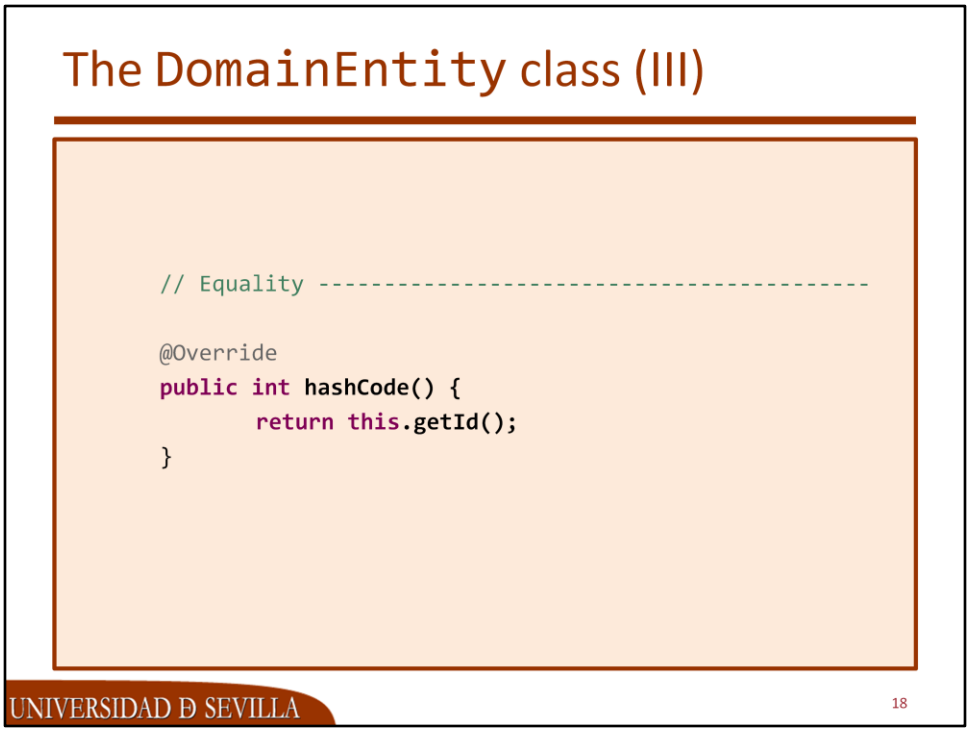
## The DomainEntity class (II)

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
public int getId() {
        return id;
}

public void setId(int id) {
        this.id = id;
}

@Version
public int getVersion() {
        return version;
}

public void setVersion(int version) {
        this.version = version;
}
```



This excerpt presents the “id” and “version” getter and setter methods:

- We use annotations “@Id” and “@GeneratedValue(strategy = Generation.TABLE)” in front of the “getId” method. The former helps JPA know which attribute in the class stores the identification of each entity; the latter instructs JPA to generate the id automatically by means of a strategy known as table strategy. The key idea is that when you create an object in memory, it gets id zero; it’s only when you persist it that JPA assigns an actual id to that object. Please, consult the bibliography to learn about other strategies to generate ids. Unfortunately, this is the only that is supported by the technology that we’re using in this subject; but this is not a limitation, it performs very well in practice.
- We use annotation “@Version” in front of the “getVersion” method. This helps JPA know which attribute in the class stores the version number of an entity.




So far you know that class “DomainEntity” endows every entity with an “id” attribute that identifies it uniquely; that means that we need to override the “equals” method. But before that, please, remember that every time you override “equals” you must also override “hashCode”. This is a method that returns a hash code of an object. Pretty simple! Note that every entity has an “id” attribute that identifies it uniquely, which implies that the value of that attribute is a perfect hash code. You should know it from your lectures on Fundamentals of Programming; if you need a tutorial to refresh your mind, please, take a look at http://www.artima.com/lejava/articles/equality.html, for instance.

**WARNING:** in the bibliography, it’s common that the “hashCode” method is overridden to compute complex black-art formulae. These formulae may work well in other contexts; in the context of domain entities, their id is a perfect hash, no other formula can work better than the id.

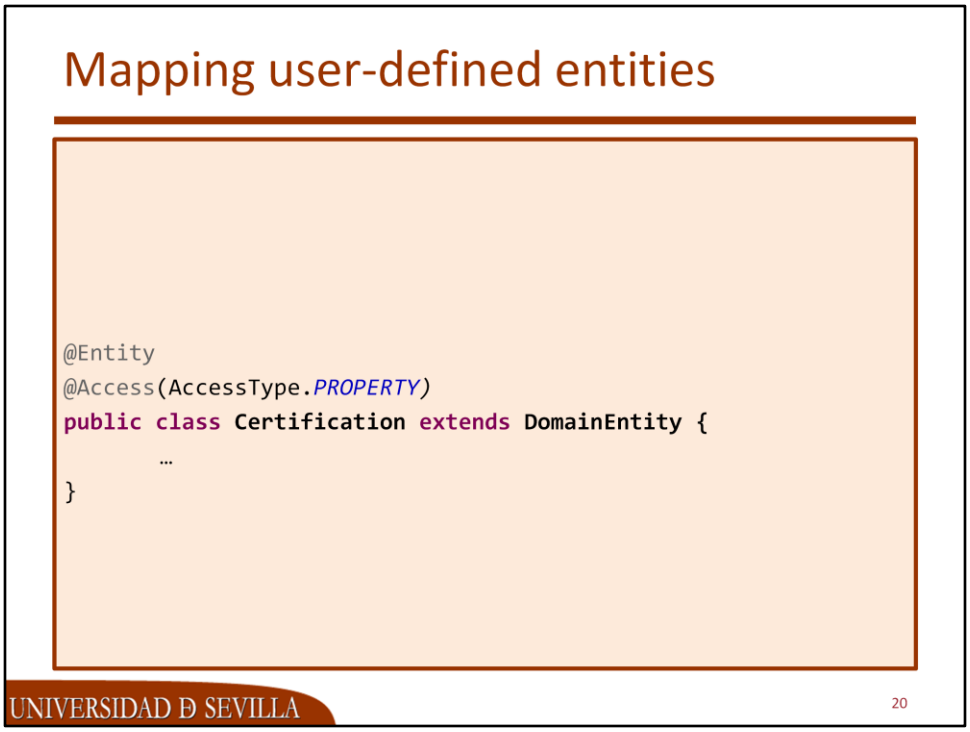
## The DomainEntity class (IV)

```
        @Override
        public boolean equals(Object other) {
            boolean result;

            if (this == other)
                result = true;
            else if (other == null)
                result = false;
            else if (other instanceof Integer)
                result = (this.getId() == (Integer) other);
            else if (!this.getClass().isInstance(other))
                result = false;
            else
                result = (this.getId() == ((DomainEntity) other).getId());

            return result;
        }

}
```

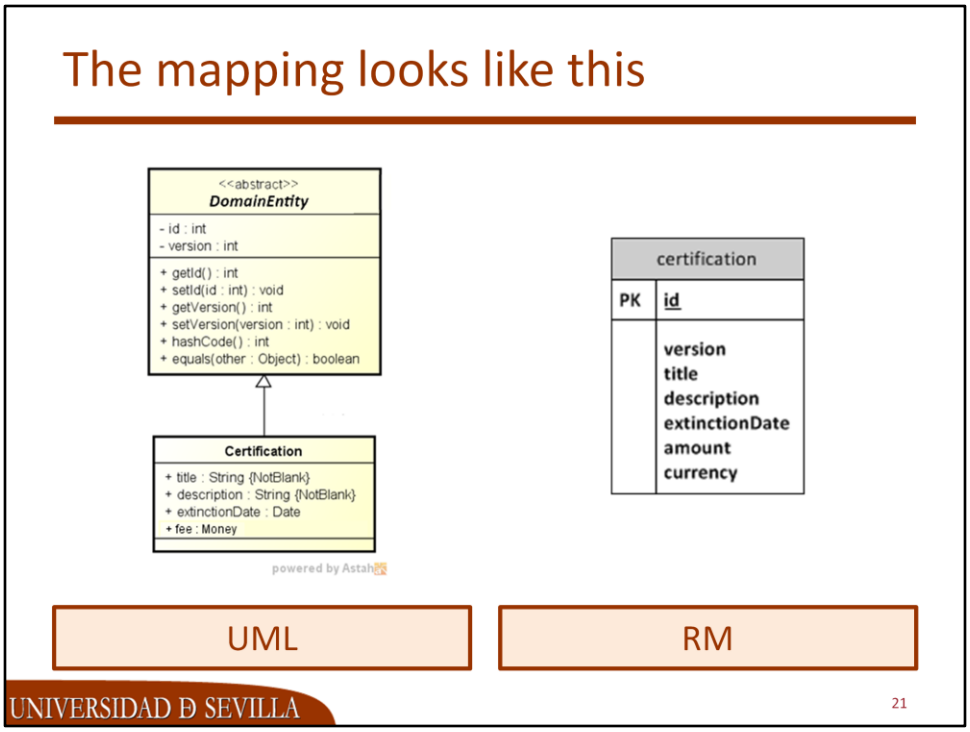


It's now time to override the "equals" method. Please, go through this code carefully. The first two branches of the if statement are trivial. The third branch is a little tricky: an object of type "Integer" can be equal to your domain object, even if that object has nothing to do with an integer. The reason why, is that the framework that we are going to use to implement our web information systems sometimes compares a domain object with an integer and expects the "equals" method to return true if the integer equals the id of the object. The fourth branch checks if the other object isn't an instance of the class of the object that executes the "equals" method; in such cases, the method trivially returns "false"; otherwise, it casts the other object to class "DomainEntity" and checks if the ids are equal or not.

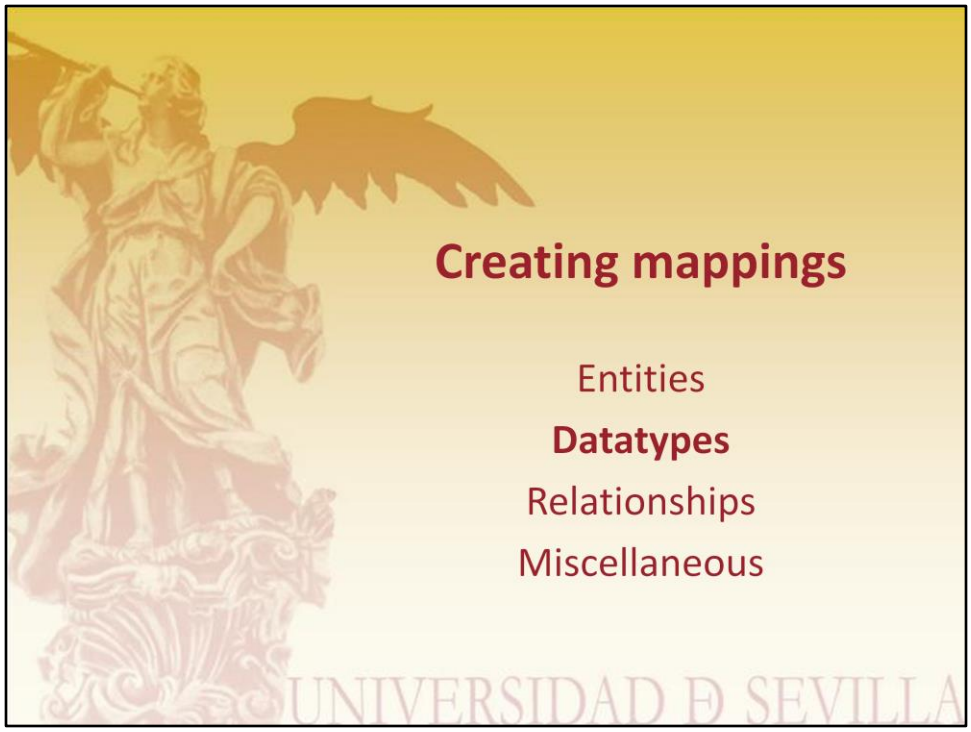
**WARNING:** should we compare version numbers, too? Please, discuss it with your partners. (The answer's no, but it's not easy to explain the reason why. Just jot this down: we are comparing object identities, not the value of their attributes; two objects with the same identity are the same, independently from their versions).




Thanks to class “DomainEntity”, mapping user-defined entities is very simple: you just need to extend class “DomainEntity” and add annotations “@Entity” and “@Access(AccessType.PROPERTY)” in front of your class declaration. Please, take into account that annotations aren’t inherited in Java; forgetting these annotations in front of a class declaration is quite a common mistake.

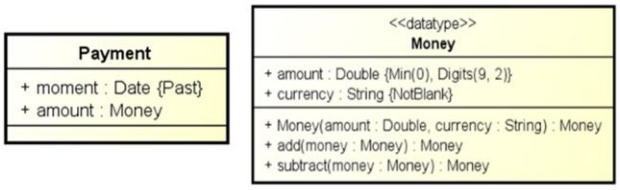



We've been talking about mapping entities for a few minutes. It's time to ask: what these mappings look like? We present an example in this slide. It's class "Certification", which is characterised by means of a title, a description, an extinction date, a fee, and, don't forget that, an inherited "id" attribute and an inherited "version" attribute. JPA maps this class onto the relational table on the right (we use Relational Model, or RM for short, to refer to the database model). Note that the "id" attribute becomes the primary key (PK) of the table, and that the remaining attributes become mandatory columns. (The custom is to typeset mandatory columns in boldface.)




What about datatypes? Let's explore them.

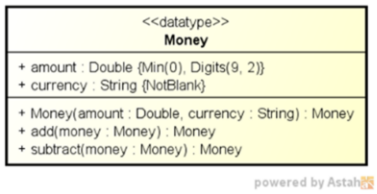
# A sample datatype






So far, you should know that a datatype is a class that represents objects that don't have an identity. In this slide, we show the only datatype that we're using in our running example. It's a class that represents amounts of money, e.g., "100 €", where "100" is the amount and "€" is the currency.

## Mapping datatypes




```
@Embeddable
@Access(AccessType.PROPERTY)
public class Money {
    …
}
```

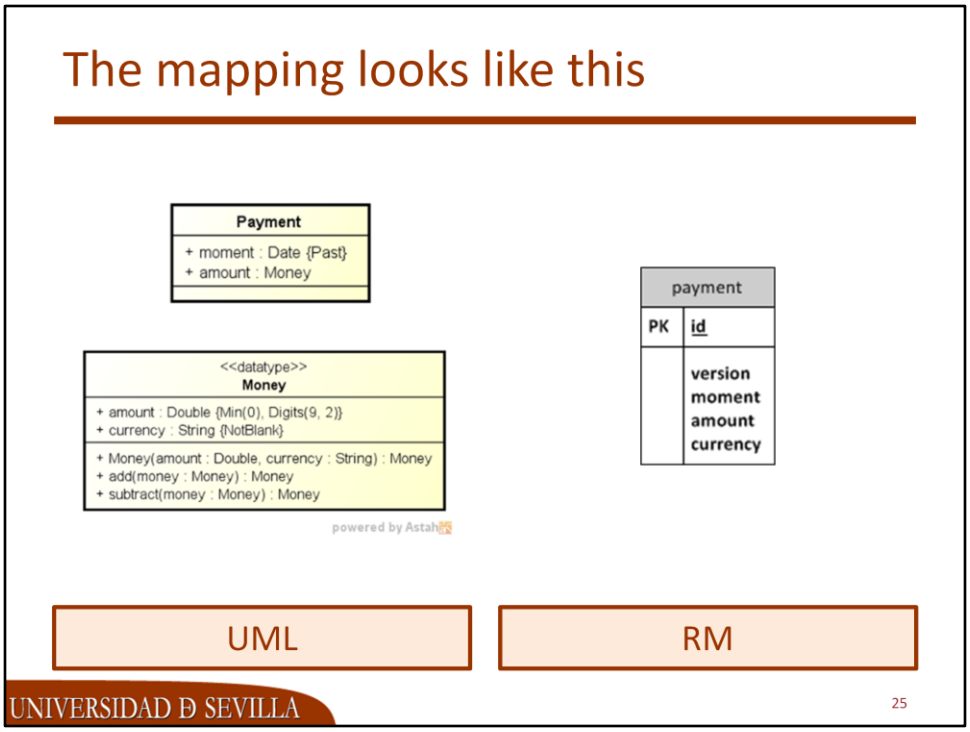
UML

Java

UNIVERSIDAD D SEVILLA

24

JPA supports datatypes by means of annotation “@Embeddable”. Note that datatypes are not entities, so you mustn’t extend class “DomainEntity” or put an “@Entity” annotation in front of the class declaration.



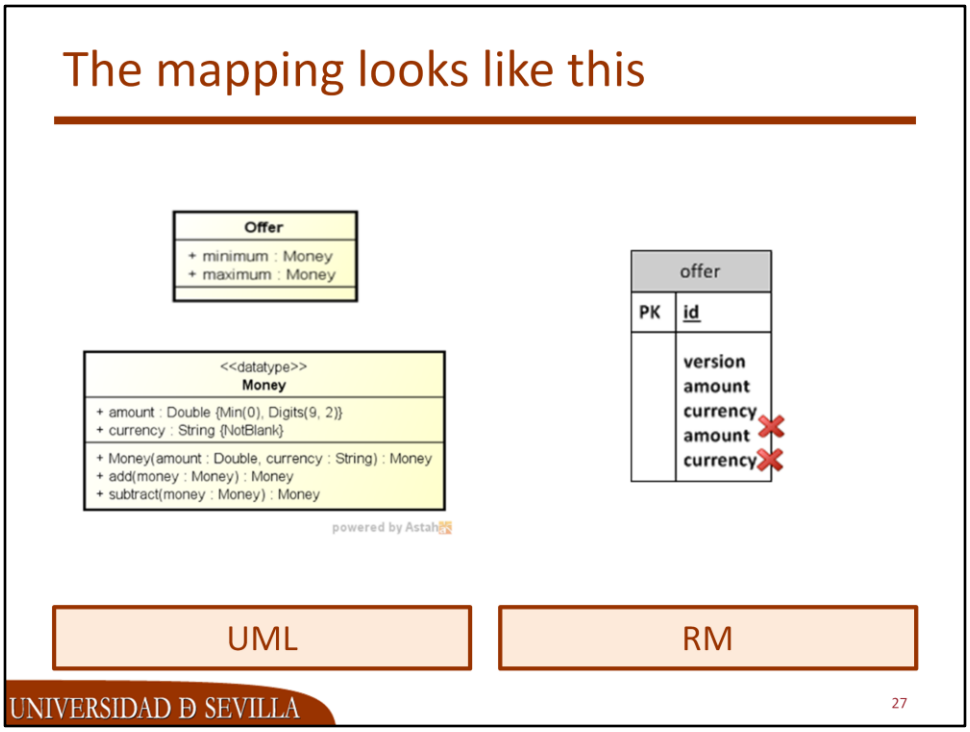

This is what the mapping of class “Payment” looks like in a relational model. Note that JPA doesn't create a new table for class “Money”, but embeds its attributes into the table that corresponds to class “Payment”.

## Hey, wait a minute!


UNIVERSIDAD Ð SEVILLA

26

Hey, wait a minute! What would happen if an entity had two attributes of type “Money”?




For instance, this slide presents a class called “Offer” that has two attributes of type “Money”. If you try to map that class, then JPA will complain since this would lead to an incorrect relational model: a table can’t have two columns with the same name.

## How to avoid name clashes?

```
@Entity
@Access(AccessType.PROPERTY)
public class Offer extends DomainEntity {

	// Attributes ----------------------------------------

	private Money minimum;
	private Money maximum;

	…
```



Fortunately, there's a simple solution to avoid name clashes, but it's cumbersome! This is an excerpt of class "Offer" in Java. Let's analyse the "getMinimum" getter.

## How to avoid name clashes?

```
@AttributeOverrides({
        @AttributeOverride(name="amount",
             column=@Column(name="minimumAmount")),
        @AttributeOverride(name="currency",
             column=@Column(name="minimumCurrency"))
})
public Money getMinimum() {
        return minimum;
}

public void setMinimum(Money minimum) {
        this.minimum = minimum;
}
```

UNIVERSIDAD D SEVILLA

29

You must use an "@AttributeOverrides" annotation, which gets an array of "@AttributeOverride" annotations as a parameter. Each "@AttributeOverride" annotation specifies the name of a property and the name of the corresponding column in the database. These annotations map property "amount" and "currency" of attribute "minimum" onto columns "minimumAmount" and "minimumCurrency" in the database.

**NOTE:** you might be tempted to write "@AttributeOverride(name="getAmount", …)", but this wouldn't work. In Java, term "property" usually refers to an attribute that has a getter method and, possibly, a setter method. It's the property that you must specify in the "@AttributeOverride" annotation, not the getter or the private field that supports the property.

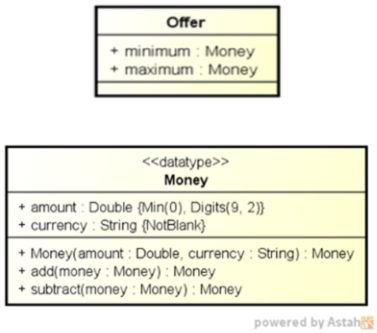
## How to avoid name clashes?

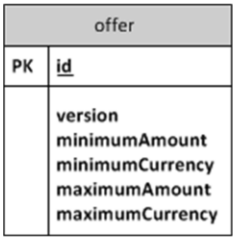
```
        @AttributeOverrides({
                @AttributeOverride(name="amount",
                        column=@Column(name="maximumAmount")),
                @AttributeOverride(name="currency",
                        column=@Column(name="maximumCurrency"))
        })
        public Money getMaximum() {
                return maximum;
        }

        public void setMaximum(Money maximum) {
                this.maximum = maximum;
        }

}
```



Similarly, these annotations map property “amount” and “currency” of attribute “maximum” onto columns “maximumAmount” and “maximumCurrency” in the database.

# The mapping looks like this



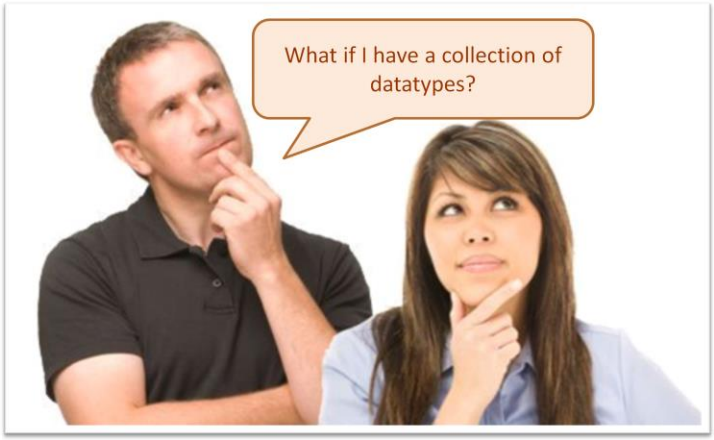

UML

RM



The resulting mapping looks like this. Too much ado about something so simple? We agree with you, but this is how technology works and you must learn it.

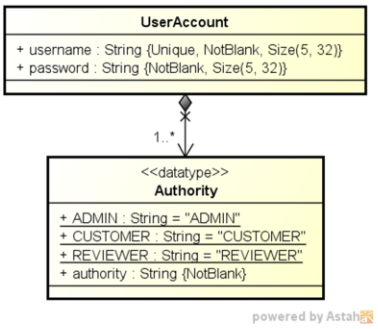
## Hey, wait another minute!






Hey, wait another minute! What if I have a collection of datatypes? That's definitely a good question, too.

# Collections of datatypes



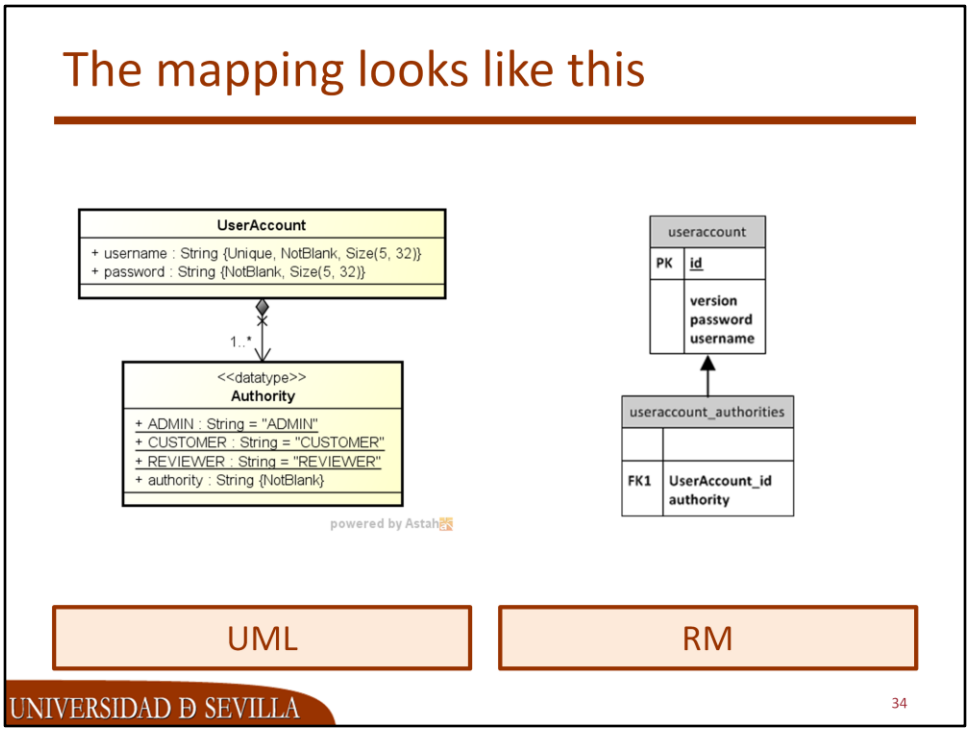

```
@ElementCollection
@Override
public Collection<Authority>
       getAuthorities() {
  return authorities;
}
```

UML

Java



In this slide, we show that objects of class “UserAccount” compose a number of “Authority” instances, and “Authority” is a datatype. That means that class “UserAccount” must have a role attribute called “authorities” of type “Collection<Authority>”. Note that we need to add annotation “@ElementCollection” to the corresponding getter method so that JPA can map it properly. (Forget about the “@Override” annotation. It’s required only in this case because method “getAuthorities” comes from a Spring interface; it’s nothing to do with persistence.)




The mapping looks like this. As expected, there's a table called "useraccount" that stores the attributes of class "UserAccount". Note that the collection of authorities that is associated with every user account is modelled as an independent table called "useraccount_authorities", and that this table doesn't have a primary key, but a foreign key called "UserAccount_id". Although not having a primary key might be a little surprising to you, this is how JPA works.

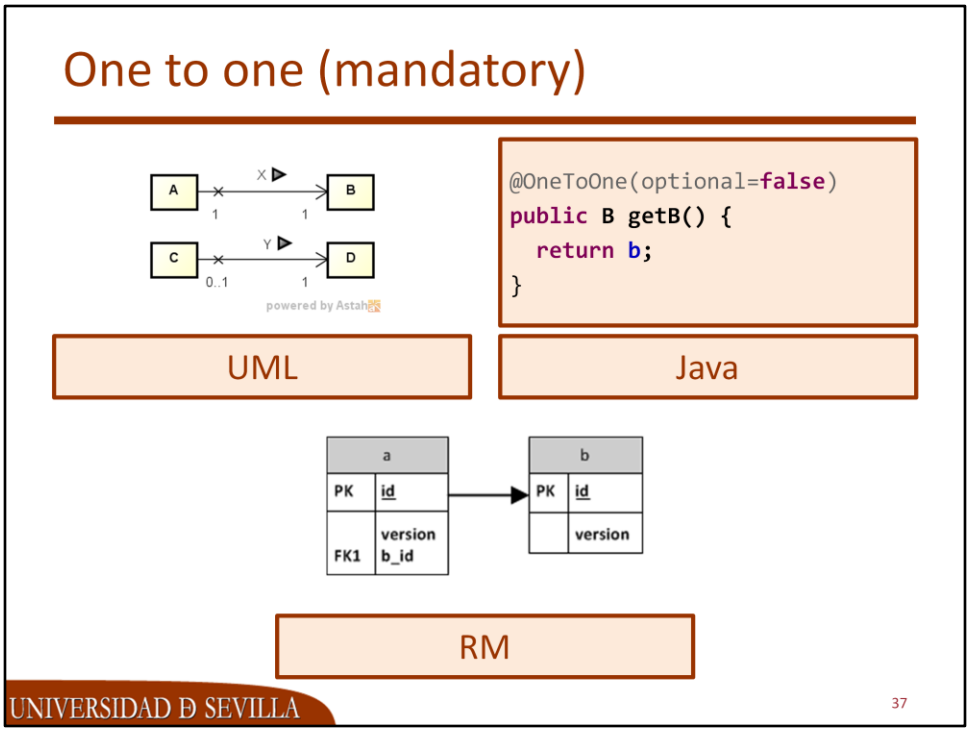

That's enough about entities and datatypes. Let's now delve into the many intricacies of relationships.

## Too many kinds of relationships

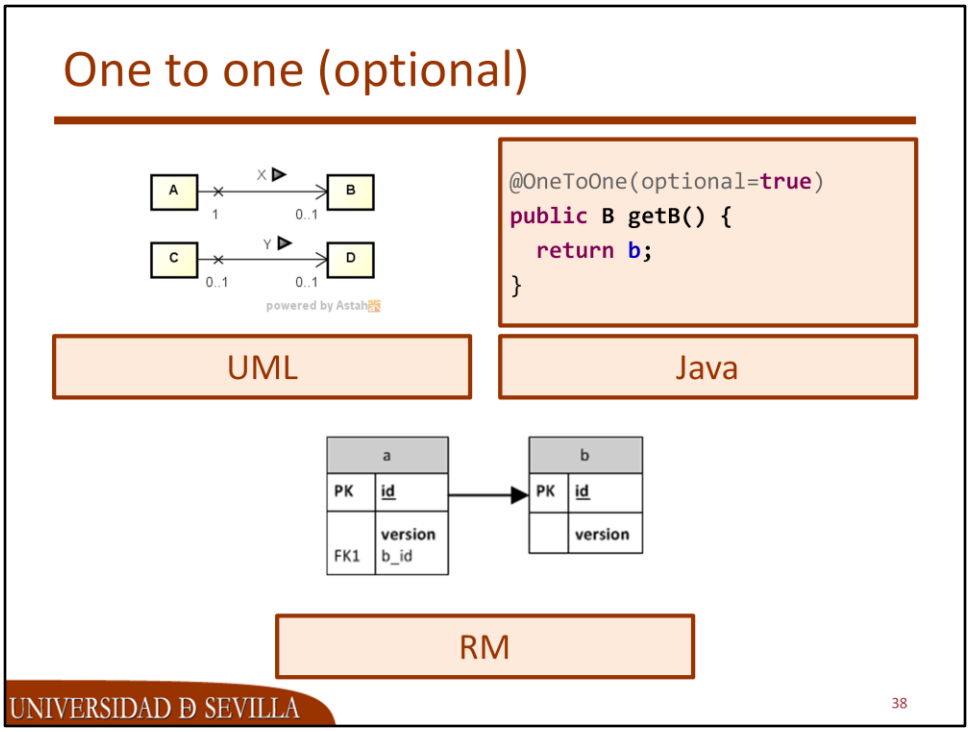
UNIVERSIDAD Ð SEVILLA

36

The only problem regarding relationships is that we need to analyse a lot of different cases, but they are quite simple in general. Let's start with unidirectional relationships.

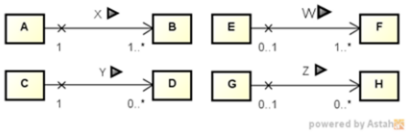



Unidirectional one-to-one mandatory associations like the ones in the upper-left box of this slide are the simplest ones. In Java, you represent them by means of an annotation of the form "@OneToOne"; note that you need to add attribute "optional=false" to make sure that the multiplicity is "1" and not "0..1", which is the default. In the database, the mapping looks like in the bottom box: the table that represents the source entity has a foreign key to the table that represents the target entity. Simple and efficient!




Unidirectional one-to-one optional associations like the ones in the upper-left box of this slide are similar to the previous ones, the only difference being that an object of the source class may be associated with zero or one instances of the target class. In Java, you just need to pass parameter “optional=true” to annotation “@OneToOne”. (This is the default value of the parameter, so passing this parameter would amount to scaffolding the annotation; the problem’s that it’s not intuitive at all, so we tolerate scaffolding this parameter.) The relational model is also very similar; note that the only difference is that the foreign key from the source to the target class isn't in boldface, which means that it’s not mandatory, that is, “b_id” may have a “null” as value.

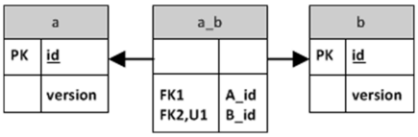
# One to many (mandatory or optional)




```
@OneToMany
public Collection<B> getBs() {
  return bs;
}
```

UML

Java

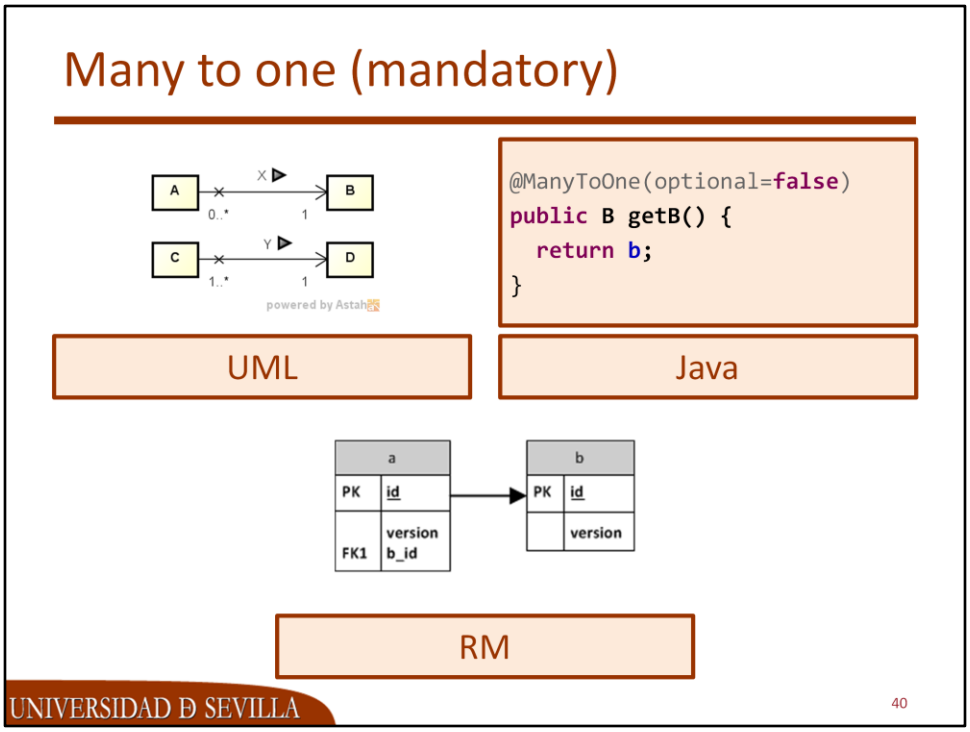



RM

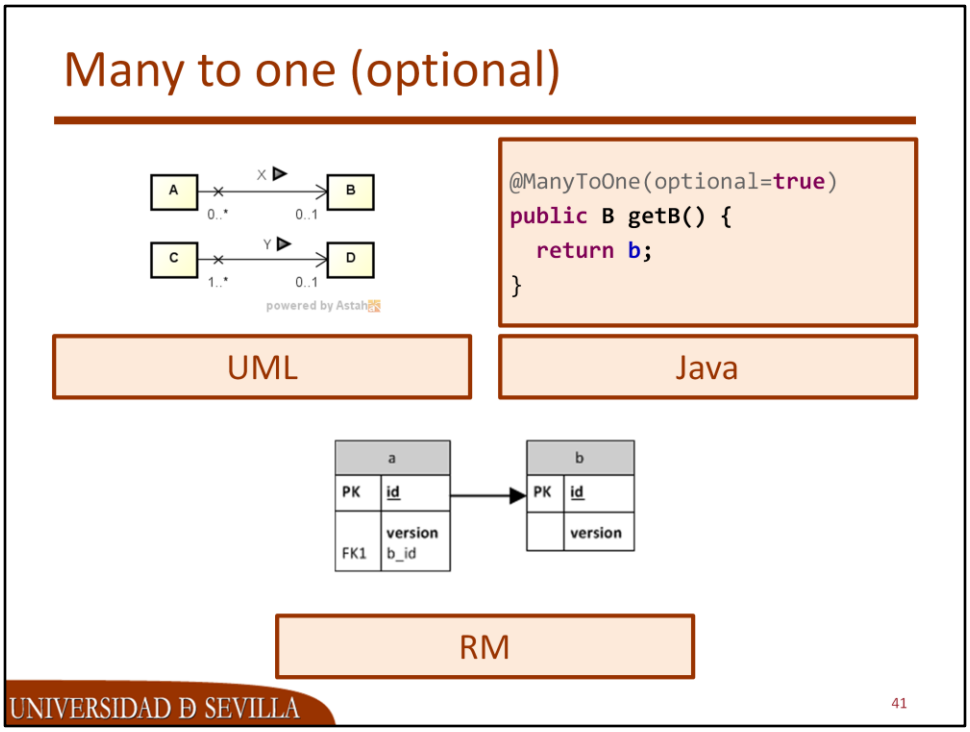
UNIVERSIDAD D SEVILLA

39

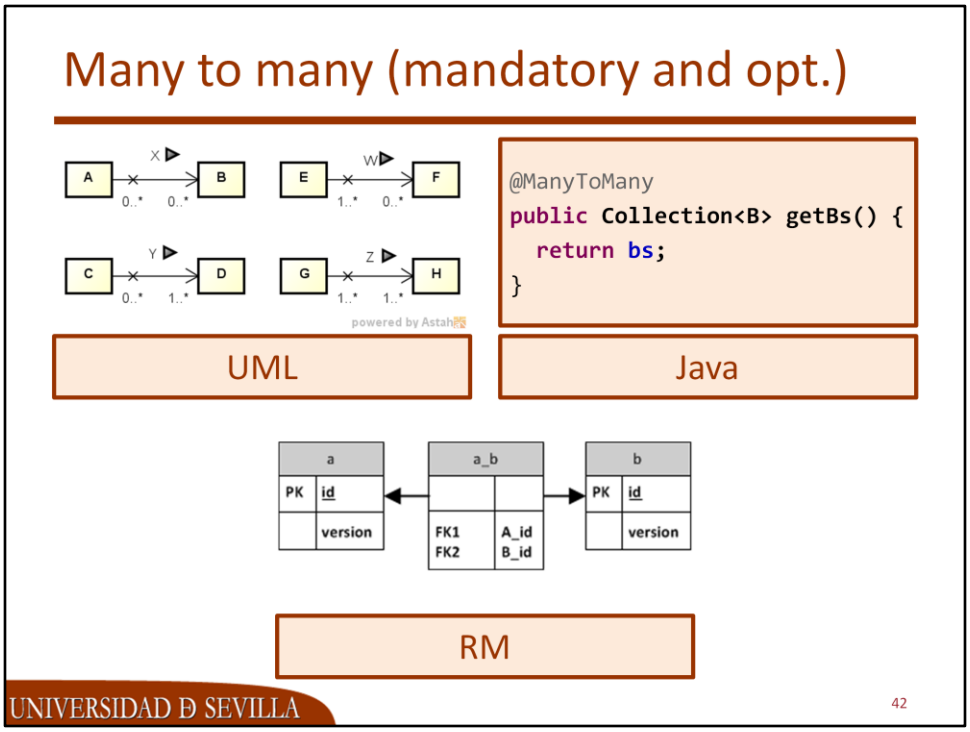
Let's now delve into unidirectional one-to-many associations, be them mandatory or optional. In this case the annotation is "@OneToMany", there's no parameter to indicate that it may be optional. Regarding the relational model, please, note that the mapping results in an intermediate table that associates instances of the source and the target classes; note, too, that this table doesn't have a primary key, but two foreign keys that refer to objects in the source and the target table; note, further, that foreign key "B_id" has a unique constraint (the "U1") which helps prevent an object of the target class from being associated to more than one object in the source class.



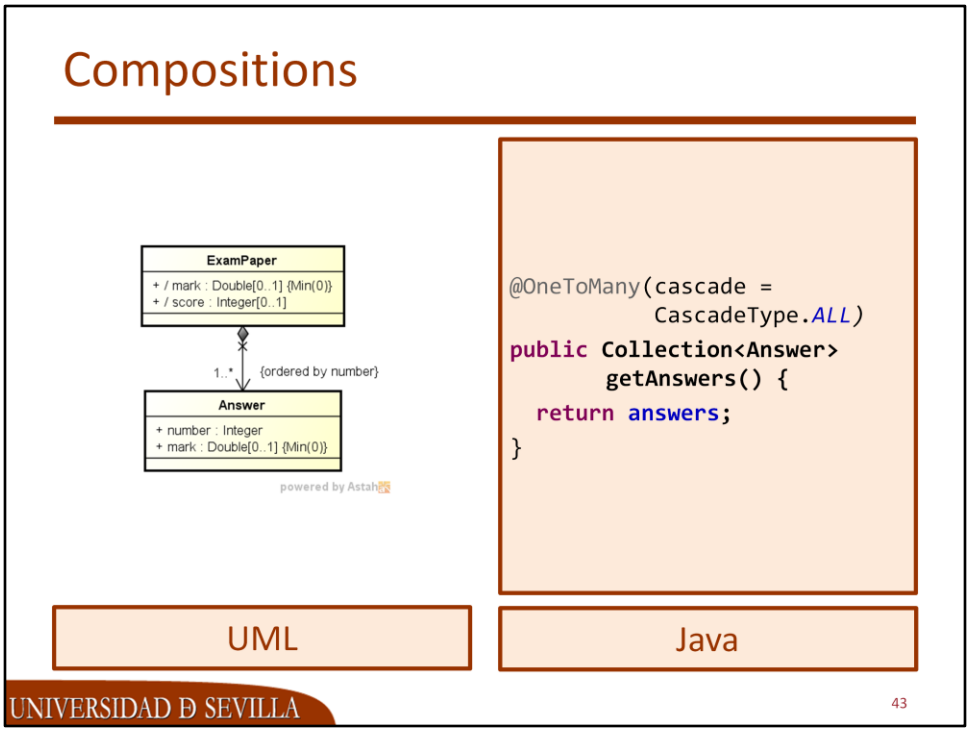

Let's now analyse unidirectional mandatory many-to-one associations like the ones in this slide; they are easy to map using the "@ManyToOne" annotation, and the resulting relational model is quite intuitive: JPA generates a table for the source class, a table for the target class, and a foreign key from the source to the target class. Pretty simple, but note that you need to add attribute "optional=false", just to ensure that every instance of the source class is associated with exactly one instance of the target class.




It's now time to review unidirectional optional many-to-one associations like the one in this slide; they are very similar to the previous ones, the only difference being that we need to pass parameter "optional=true" on to annotation "@ManyToOne" and that the resulting foreign key in the relational model is nullable (note that it's not in boldface). Note that "true" is the default value for this parameter, so you may omit it if you wish; we however tolerate scaffolding it because it's not that intuitive. Pretty simple, as well.



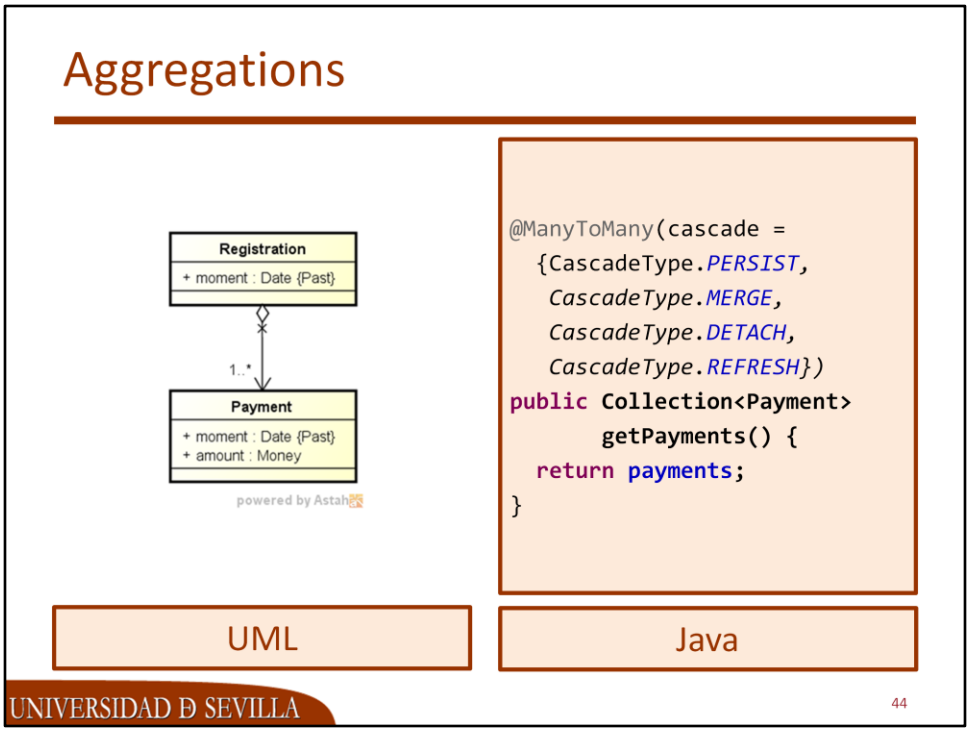

There are also many unidirectional many-to-many associations like the ones in this slide. They are dealt with using annotation "@ManyToMany"; the mapping to the database looks like in this slide: the source and target entities are mapped to their corresponding tables, and the association itself is mapped onto an intermediate table that stores references to the pairs of objects that are associated. Realise that this table doesn't have a primary key, which shouldn't be surprising to you now.




Let's now analyse compositions. Intuitively, a composition is a one-to-many association in which the lifecycle of the part depends on the whole. You've read that several times, but chances are that you haven't understood it. You know that the "whole" refers to the root of the composition (class "ExamPaper" in our example) and the "part" refers to the leaves of the composition (class "Answer" in our example), right? The fact that the lifecycle of the part depends on the whole means that whenever you persist an "ExamPaper" object to the database, the corresponding "Answer" objects must be persisted as well; you don't need to persist each "Answer" object independently, it's JPA that must care of that; similarly, if you delete an exam paper, you expect the answers to be deleted as well, and so on. This behaviour is known as cascading and it's specified by means of parameter "cascade=CascadeType.ALL" in the corresponding annotation.

**NOTE:** Please, note that the navigation pattern in this example cannot be implemented efficiently, since it's a unidirectional one-to-many association. It's intended only for illustration purposes. In a few slides, we'll report on bidirectional relationships.

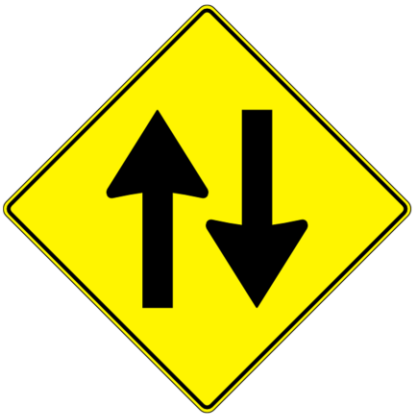
**WARNING:** "CascadeType" is an enumerated type that is defined in several packages. Please, use the type defined in package "javax.persistence". Otherwise you'll get an error that is very difficult to understand: "Type mismatch: cannot convert from CascadeType to CascadeType[]".




Aggregations are very similar to compositions, the difference being that whenever you delete the whole, the parts remain; furthermore, a part may be in several aggregations. IN this case, the annotation is "@ManyToMany", but the parameter differs in that we need to specify that all operations except for "CascadeType.REMOVE" must be cascaded. We're pretty sure you know little about operations "CascadeType.MERGE", "CascadeType.DETACH", and "CascadeType.REFRESH"; don't worry about them, they are of little interest to us. It's very unlikely that you need to use them in practice, which is the reason why we won't provide any additional details now; if you're interested, please take a look at the bibliography that we recommend at the end of this lecture.

**NOTE:** Please, note that the navigation pattern in this example cannot be implemented efficiently, since it's a unidirectional one-to-many association. It's intended only for illustration purposes. In a few slides, we'll report on bidirectional relationships.

## What ’bout bidirectional relationships?

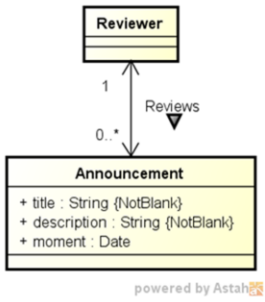


UNIVERSIDAD Ð SEVILLA

45

So far, we’ve only studied unidirectional relationships. What about bidirectional ones? Bidirectional relationships can be seen as if they were two related unidirectional relationships. Obviously, the meaning isn't the same. A bidirectional relationship is one relationship; two unidirectional relationships that simulate a bidirectional one are two different relationships, not one; that means that you may easily get in trouble if you forget to update these two different relationships co-ordinately.

## Use mappedBy on one side…




```
public class Announcement {
  …
  @ManyToOne(optional=false)
  public Reviewer
    getReviewer() {
      return reviewer;
  }
  …
}
```

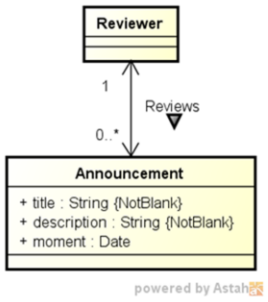
UML | Java

UNIVERSIDAD D SEVILLA

46

For instance, this example shows an excerpt of our sample model in which we specify that a reviewer reviews an arbitrary number of announcements (possible none), and that an announcement is reviewed by exclusively one reviewer; furthermore, we specify that the association is navigable in both directions. In Java, we need to implement this association as follows: there's a role attribute called "reviewer" in class "Announcement", which is of type "Reviewer"; we then put an "@ManyToOne" annotation in front of the corresponding getter since the association is many to one if we read it from the perspective of class "Announcement".

## … and on the other side, too




```java
public class Reviewer {
  …
  @OneToMany(mappedBy="reviewer")
  public Collection<Announcement>
  getAnnouncements() {
    return announcements;
  }
  …
}
```

UML | Java

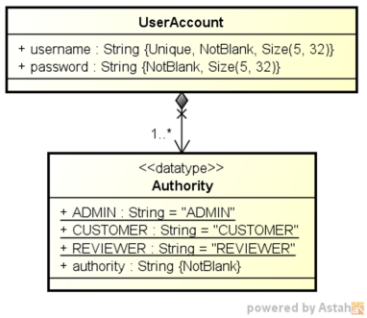
UNIVERSIDAD D SEVILLA

47

Similarly, class “Reviewer” has a role attribute called “announcements”, which is of type “Collection<Announcement>”. From the “Reviewer” class, this association is a one-to-many association, which is the reason why we put annotation “@OneToMany” in front of the “getAnnouncements” getter. But note that we have to use an additional parameter in this case: “mappedBy="reviewer"”. This parameter helps JPA know that the “announcements” role attribute in class “Reviewer” is related to the “reviewer” role attribute in class “Announcement”; this prevents JPA from dealing with these role attributes as if they were two independent associations. Please, bear in mind that JPA just cares of persistence; it’s your responsibility to keep the values of both attribute roles in sync. That is, whenever you set the reviewer of an announcement, it’s your responsibility to add that announcement to the list of announcements of the corresponding reviewer. JPA won’t add it for you!


Ufff…. too many slides regarding relationships, don't you think so? Unfortunately, we need to command them all so that we can easily and efficiently persists our Java domain models. Before concluding, let's take a look at a few miscellaneous annotations.

# Single unique attributes




```
@Column(unique = true)
public String getUsername() {
    return username;
}
```

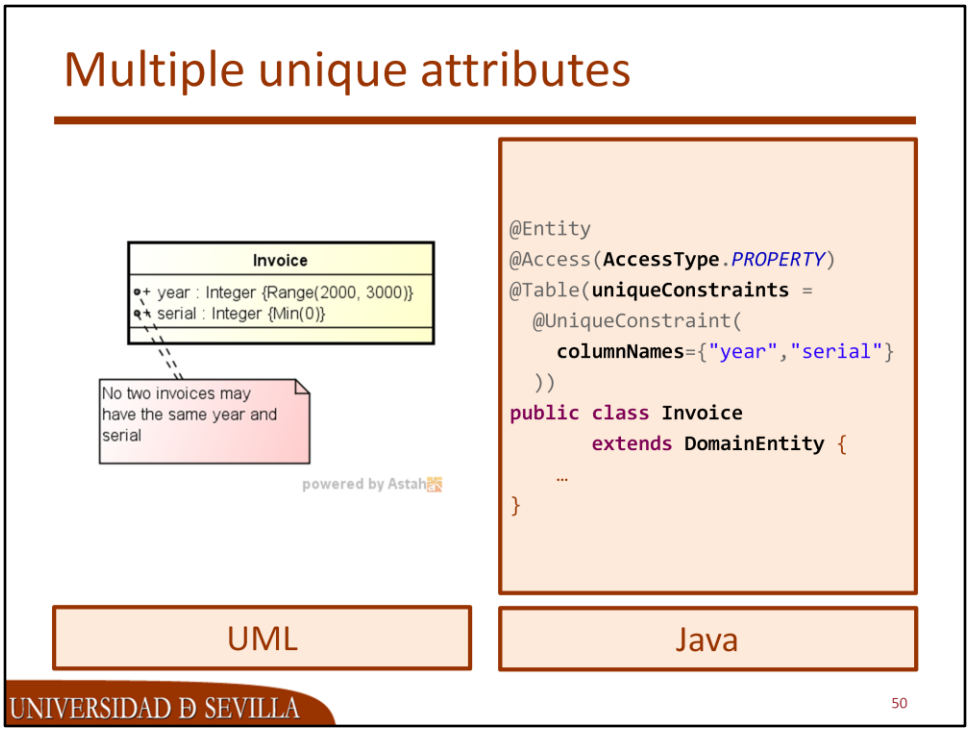
UML

Java

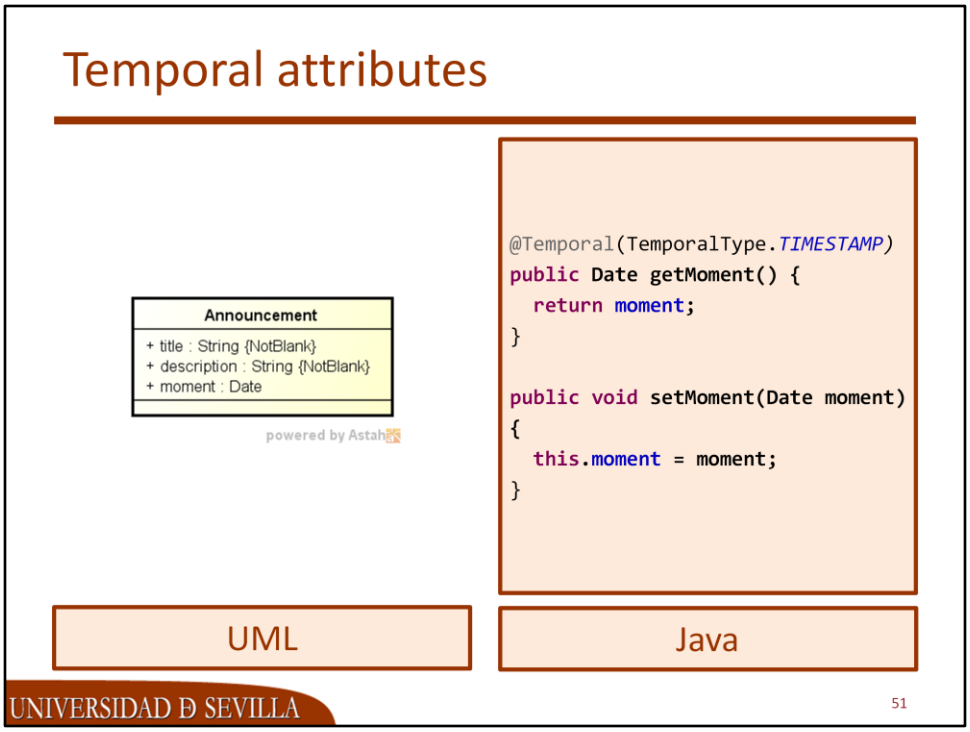
UNIVERSIDAD D SEVILLA

49

Can you remember class “UserAccount”? We’ve used it as an example several times. By now, you should know that this class represents user accounts, which are characterised by a username and a password. Note that a username mustn’t be blank, its size must be from 5 to 32 characters… but there’s an additional constraint to which we haven’t paid a lot of attention so far: “Unique”. As you may easily expect, a “Unique” constraint requires that no two objects of a class have the same value for the corresponding attribute; in this example, no two user accounts may have the same username. This constraint is implemented as annotation “@Column(unique=true)”.

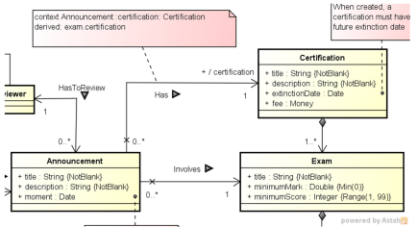



Although it is not the case in our running example, it's quite common in practice that multiple attributes have to be unique together, not independently. For instance, think of a class called "Invoice"; it's not uncommon that invoices are identified by means of a code that consists of a year number and a serial number, e.g., "2010-00000" or "2013-12345"; what matters in this example is that no two different invoices may have the same code, even though many invoices have the same year or the same serial number. To specify such multiple unique attributes, you must use a user-defined constraint in UML; in JPA, your must use a "@Table" annotation whose syntax is shown in this slide.




So far we've added annotations to role attributes only. JPA is able to deal with attributes of basic types like integers, doubles, booleans, characters, strings, and so on. There's, however, a datatype with which it can't deal without a little help from the user: "Date" from package "java.util". In spite of what its name might suggest to you, an instance of "Date" stores both a date and a time. That means that you may use an attribute of type "Date" to store a date, a time, or both… but JPA doesn't know what your choice was. That's why you have to use annotation "@Temporal(TemporalType.DATE)" to indicate that you're storing just a date, "@Temporal(TemporalType.TIME)" to indicate that you're storing just a time, or "@Temporal(TemporalType.TIMESTAMP)" to indicate that you're storing both a date and a time.

# Transient derived attributes




```java
@Transient
public Certification
       getCertification() {
  Certification result;

  if (exam != null)
    result = exam.getCertification();
  else
    result = certification;

  return result;
}

public void setCertification
     (Certification certification) {
  this.certification = certification;
}
```

UML

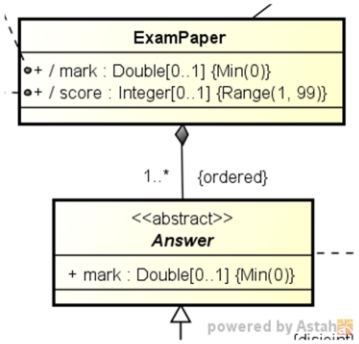
Java

UNIVERSIDAD D SEVILLA

52

Derived attributes that can be computed on-the-fly must be implemented using the "@Transient" annotation, which indicates JPA that their value does not need to be persisted to the database, but will be computed by the corresponding getter methods. Think for instance of the derived role attribute called "certification" in class "Announcement"; in a previous lecture, we justified that this derived attribute makes sense as long as it may ease our programmers' lives. Computing the value of such an attribute is very easy, so it makes sense to put a "@Transient" annotation in front of the corresponding getter.

## Persistent derived attributes




```
public Double getMark() {
  return mark;
}

public void setMark(Double mark) {
  this.mark = mark;
}

public Integer getScore() {
  return score;
}

public void setScore(Integer score)
{
  this.score = score;
}
```

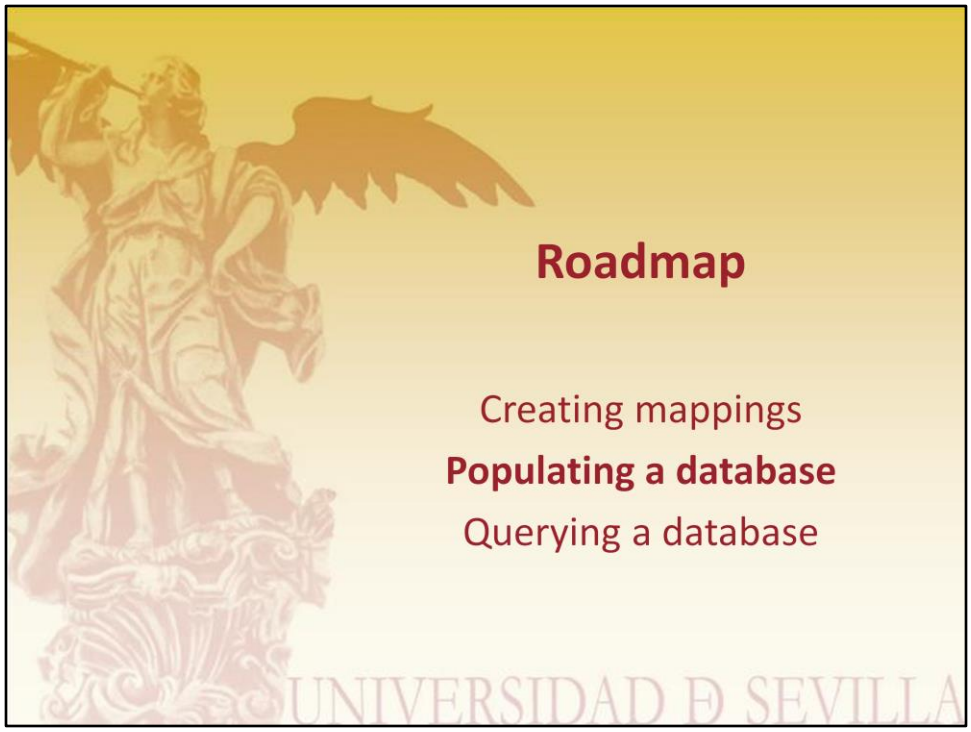
UML

Java

UNIVERSIDAD D SEVILLA

53

Class “ExamPaper”, on the contrary, provides two good examples of derived attributes that have to be persisted or, otherwise, computing them on-the-fly would be inefficient. Attribute “mark” is expected to store the mark of an exam paper; computing the value of this attribute requires retrieving every answer and summing up their individual marks; once an exam paper is submitted and marked, its mark cannot change, so it makes sense to compute it only once, not every time it has to be consulted. Attribute “score” is expected to store the percentile in which the mark of an exam paper falls, in the context of the other exam papers that were produced in the same announcement; once again, computing that value is a non-trivial task that takes time. Note that we don’t need to add any annotations in front of the getters of such derived attributes; they are persisted by JPA as if they were regular attributes whose values are assumed to be computed by means of external procedures.




That was a good review of how to create mappings. It's now time to delve into how to populate a database.

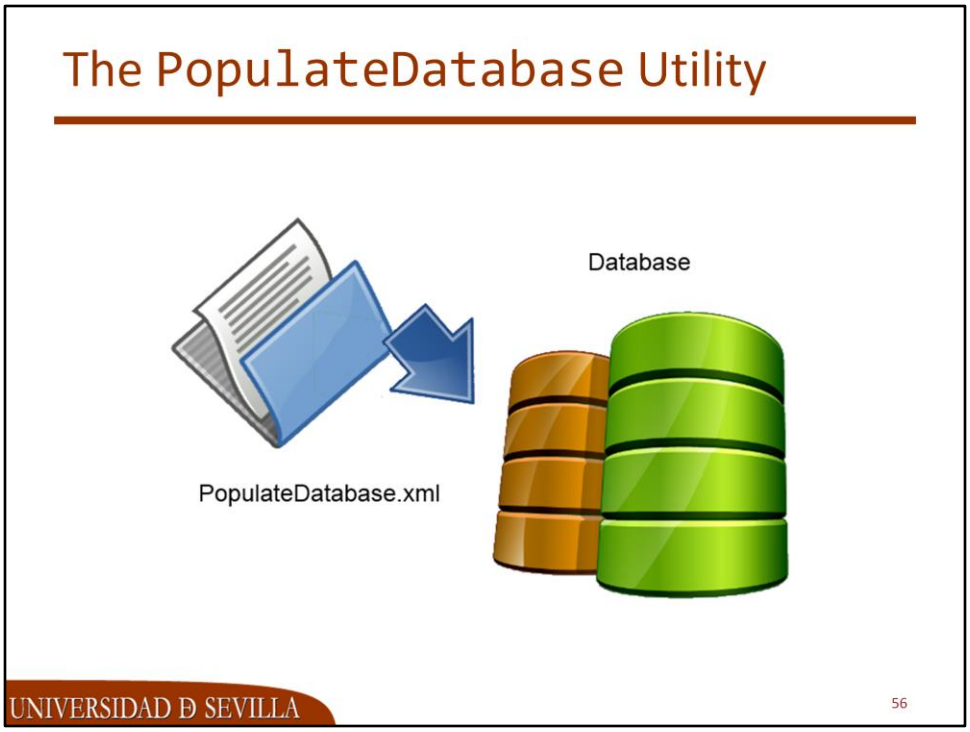
## What is it?

Populating a database means inserting data into that database

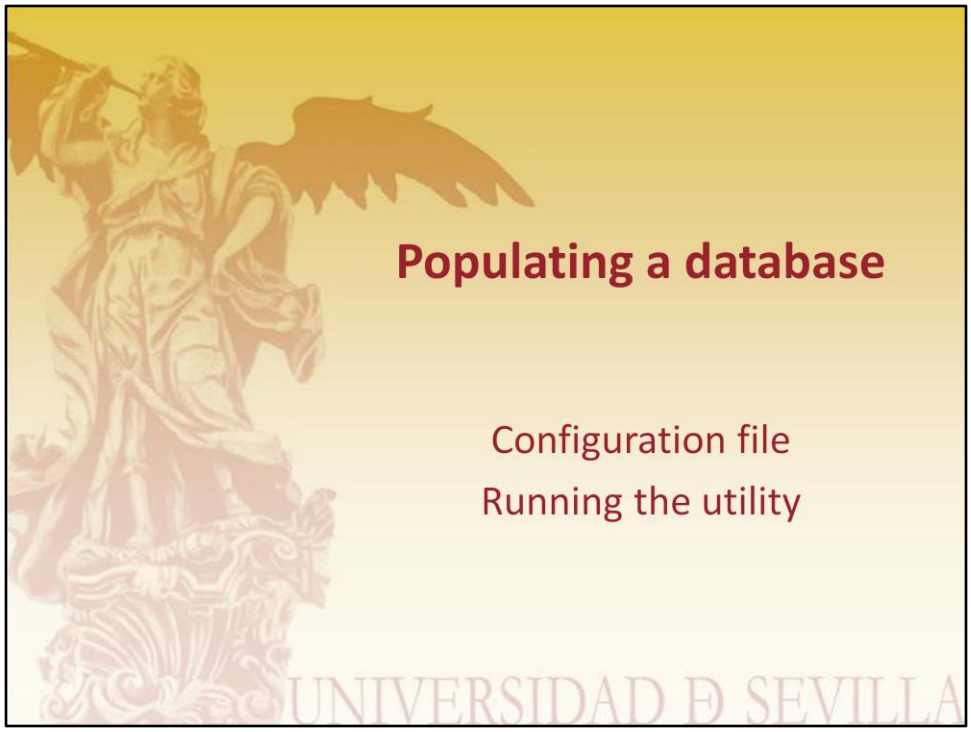
UNIVERSIDAD Ð SEVILLA

55

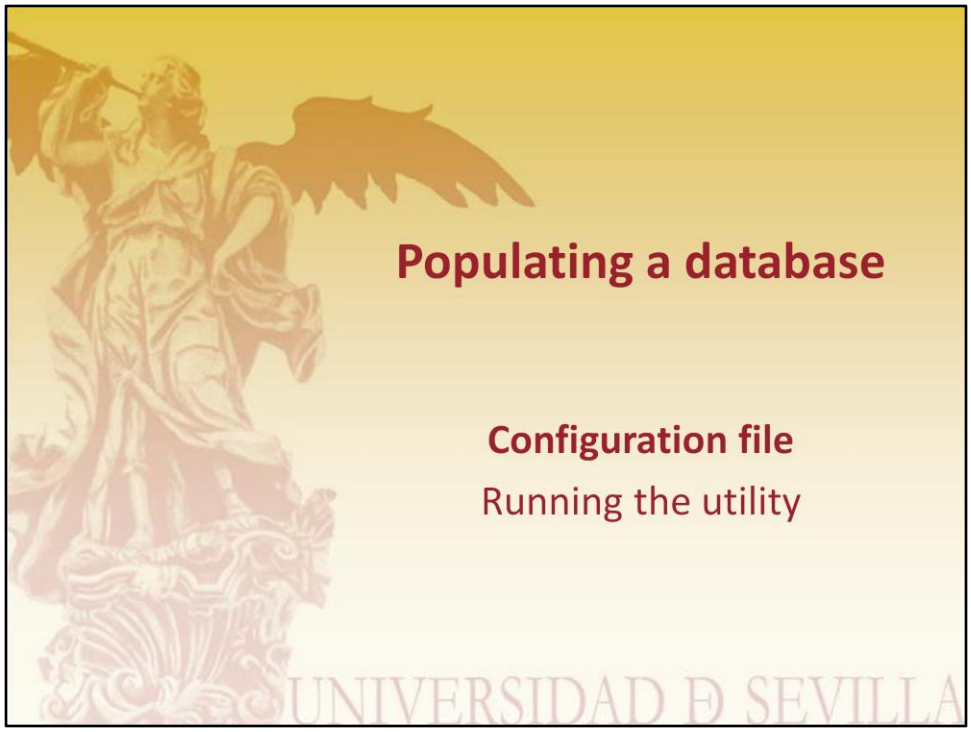
Populating a database means inserting data into that database. That shouldn't be something new to you.




In our project template, we provide a utility called “PopulateDatabase.java”. We’ve used that utility several times: in Lesson “L01 – Introduction”, we told you how to execute it to create three initial user accounts; that is, you should be familiar with it now. It’s then time to delve into the intricacies of this utility. Basically, it reads a configuration file called “PopulateDatabase.xml”, drops the corresponding database if it exists, (re-)creates it, and inserts the objects that are specified in the configuration file into the database.

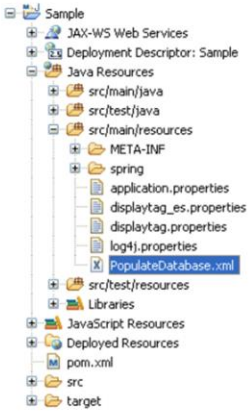



Thus, in this section, we're going to report on the configuration file and then on how to run the utility.




Let's start with the configuration file.

# The PopulateDatabase.xml file






The “PopulateDatabase.xml” file is an XML file that provides a description of a web of objects. It’s available in your project template in folder “src/main/resources”.


File “PopulateDatabase.xml” starts with this XML header that states that “http://www.springframework.org/schema/beans” is the default namespace. It provides an element called “beans”, by means of which we can specify how to construct an object.

**NOTE:** we assume that you know the foundations of an XML document. It has a header (“<?xml version="1.0" encoding="UTF-8"?>”) and then comes a single element that provides schema information. In the example, that element is “beans”, and the schema information is provided in attribute “xmlns”; this attribute indicates that the definition of the “beans” element is available in namespace “http://www.springframework.org/schema/beans”. A namespace is like a Java package: a logical container where someone puts a number of element definitions; there’s a default namespace (the one introduced by means of the “xmlns” attribute); additional namespaces must be introduced by means of attributes of the form “xmlns:<name>="<namespace>"”. For instance, we introduce a namespace called “xsi” which defines an attribute called “schemaLocation”; this schema attribute allows to specify where the definition of each namespace is located (except for the “xsi” namespace itself).

## Creating a Certification object

```
<bean id="certification1" class="domain.Certification">
    <property name="title" value="Certification 1"/>
    <property name="description" value="Description 1"/>
    <property name="extinctionDate" value="01/02/2015 23:59"/>
    <property name="fee" value="100.00"/>
    <property name="exams">
        <list>
            <ref bean="exam1"/>
            <ref bean="exam2"/>
        </list>
    </property>
</bean>
```



This is an excerpt of a “PopulateDatabase.xml” configuration file that specifies how to create a “Certification” object: “bean” is the XML element used to introduce a new object, the “id” attribute indicates the name of the object, the “class” attribute its class (note that it must be fully qualified), and then comes a list of properties, each with a “name” attribute and a “value” attribute. For instance, this specification creates a “Certification” object with title “Certification 1”, description “Description 1”, extinction date “February 1, 2015 23:59”, and fee “100.00” euro. Note that “exams” is a role attribute of type “Collection<Exam>”; that is, it stores the collection of exams that compose this certification. Collections are introduced using the “list” xml element; if it’s a collection of objects (entities or datatypes), then they are referenced by means of element “ref”; if it’s a collection of simple primitive types, then you can use element “value”, e.g., “<list><value>1.0</value><value>2.0</value></list>”. In addition to lists you can also create sets using element “<set>” and arrays using element “<array>”. If you need to set a property to null, you can use the following syntax: “<property name="…"> <null/> </property>”.

## Creating an Exam object

```
<bean id="exam1" class="domain.Exam">
    <property name="title" value="Exam 1" />
    <property name="minimumMark" value="5" />
    <property name="minimumScore" value="1" />
    <property name="certification" ref="certification1" />
    <property name="questions">
        <list>
            <ref bean="question1" />
            <ref bean="question2" />
        </list>
    </property>
</bean>
```



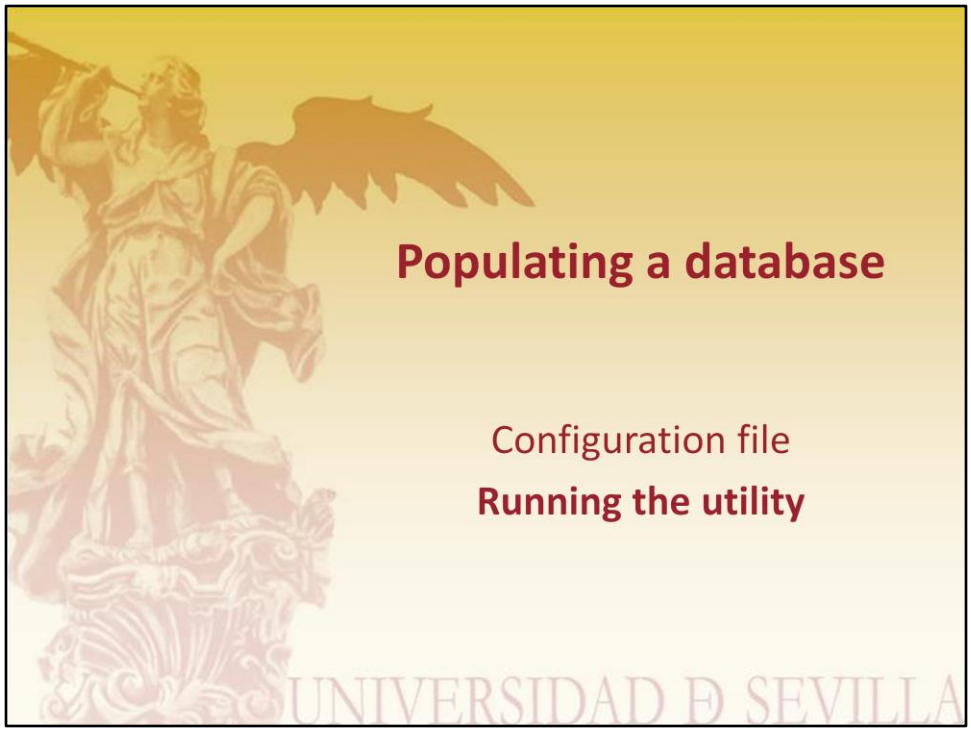
This excerpt shows how to create an “Exam” object. We don’t think you should have any problems at interpreting this bean element. We’d just like to attract you attention to role attribute “certification”; note that it references a single “Certification” object, not a collection. In such cases, you must use a “property” element with a “ref” attribute instead of a “value” attribute.

## Creating an OpenQuestion object

```
<bean id="question1" class="domain.OpenQuestion">
    <property name="number" value="1" />
    <property name="statement" value="Question 1" />
    <property name="maximumMark" value="1" />
    <property name="answer" value="Answer 1" />
    <property name="exam" ref="exam1" />
</bean>
```

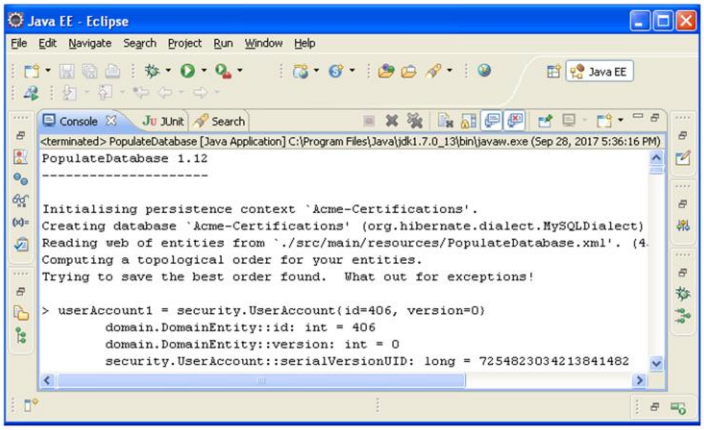


Neither should it be difficult to understand what this definition means: it's the specification of an "OpenQuestion" object whose number is "1", whose statement is "Question 1", whose maximum mark is "1" point, whose answer's "Answer 1", and it's a part of the object called "exam1".




Let's now provide a few details on how to run the population utility.

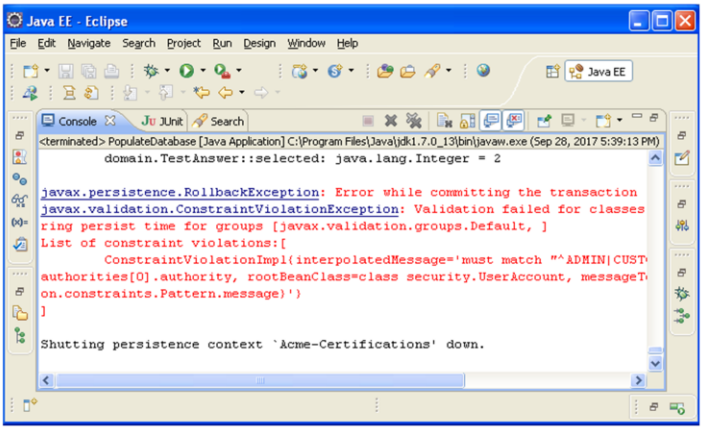
## It’s very simple





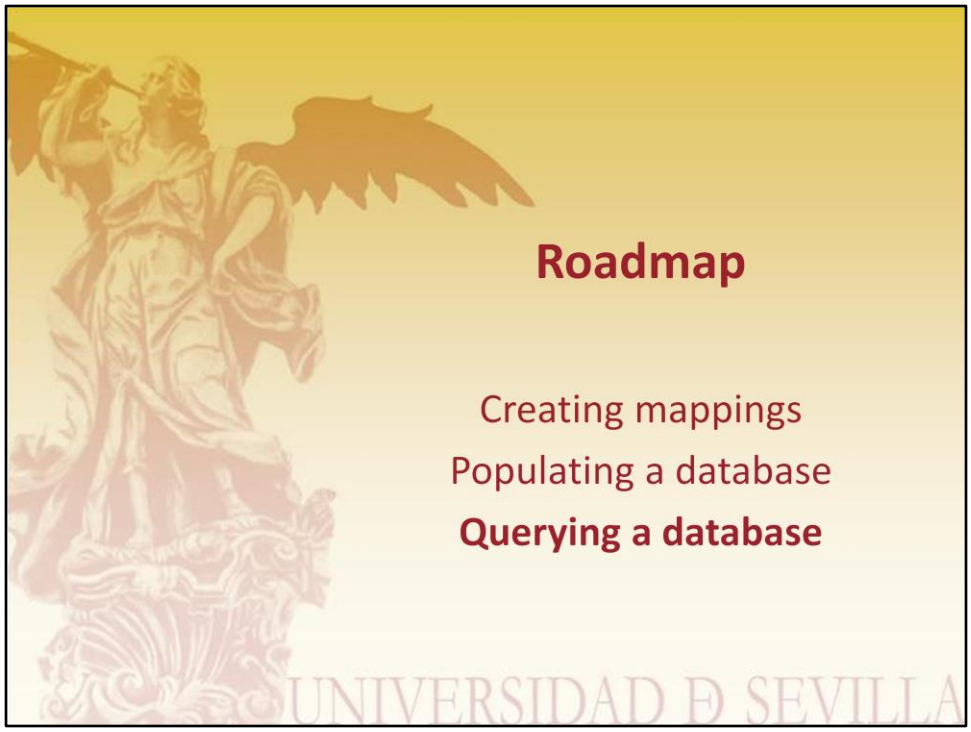

Once you have a configuration file ready, just launch the “PopulateDatabase” utility. Right click on the java file, select “Run as > Java Application”, and you should see something similar to this screenshot on your console. If everything goes well, you should see just a log with lines of the form “Persisting (id-name, class, id-integer)”, where “id-name” denotes the id you’ve declared in your configuration file, “class” the class of the corresponding object, and “id-integer” the identifier that every object is assigned when persisted.

# Watch the exceptions out!






Unfortunately, exceptions happen. The only solution to solve the majority of problems when populating a database is to create the "PopulateDatabase.xml" configuration file step by step, in tiny chunks in which you add a couple of objects, run the utility, mend eventual exceptions, and go ahead.



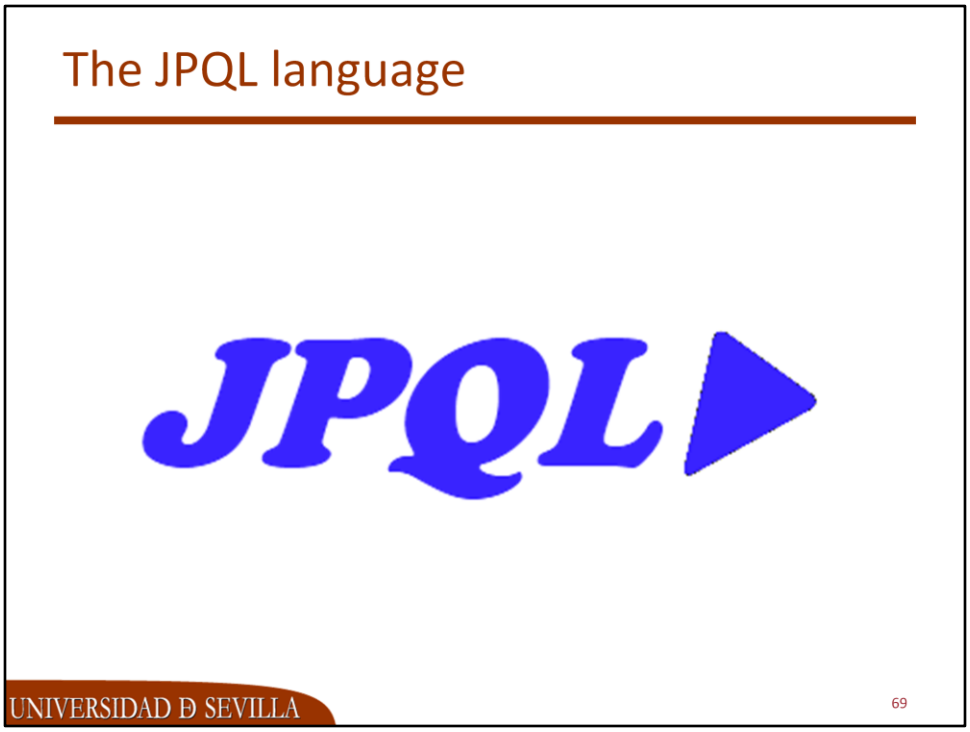

Once we know how to populate a database, let's delve into querying it.

## What is a query?

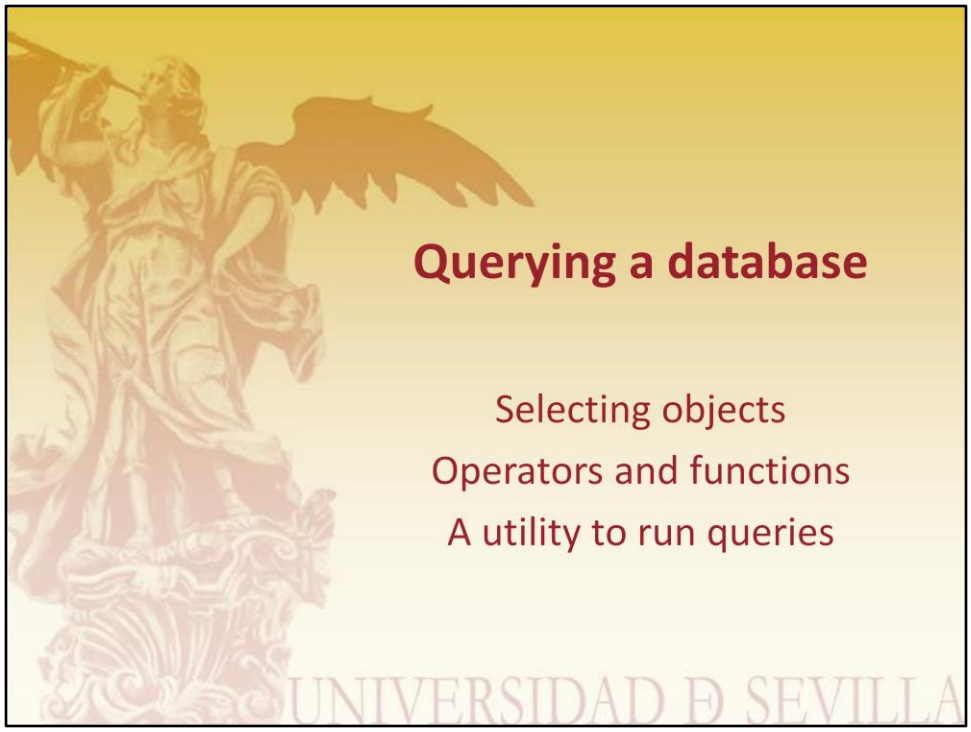
It's a statement that helps retrieve objects, insert, update, or delete them



First of all, we should know what a query is. Generally speaking, a query is statement that helps retrieve objects, insert, update, or delete them. (Note that some people find it a little odd that a statement that deletes data from a database is a query, but it is).

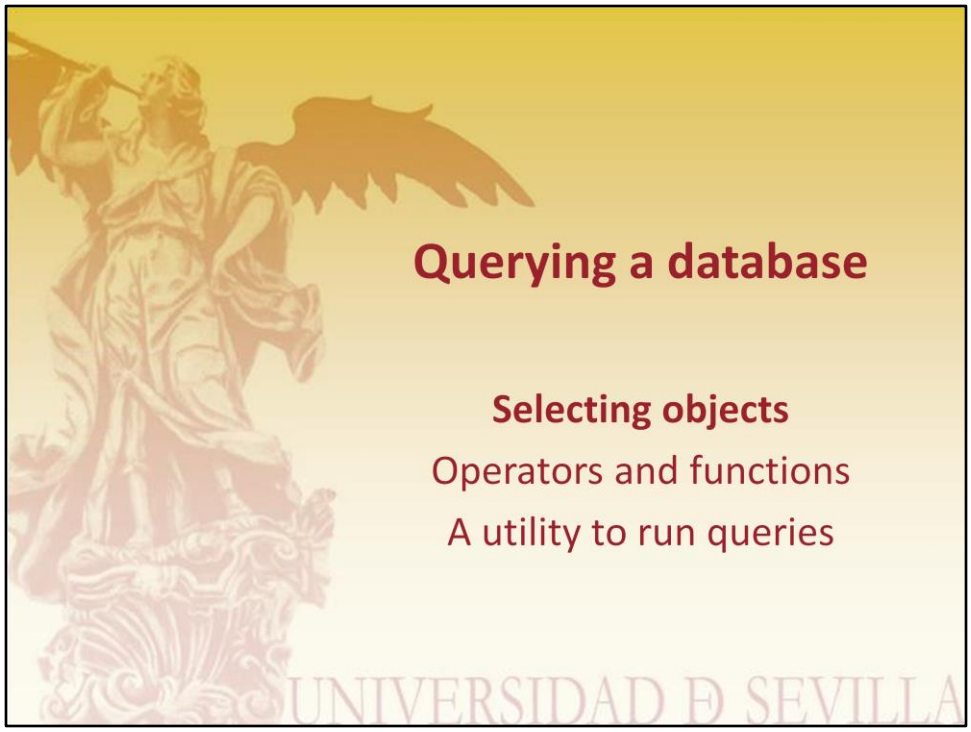



The language that we're going to use to query our databases is JPQL, which stands for Java Persistence Query Language. It's somewhat similar to SQL, which you should have learnt in previous subjects; the most important difference is that JPQL is object-oriented and that it's very portable across different database management systems.

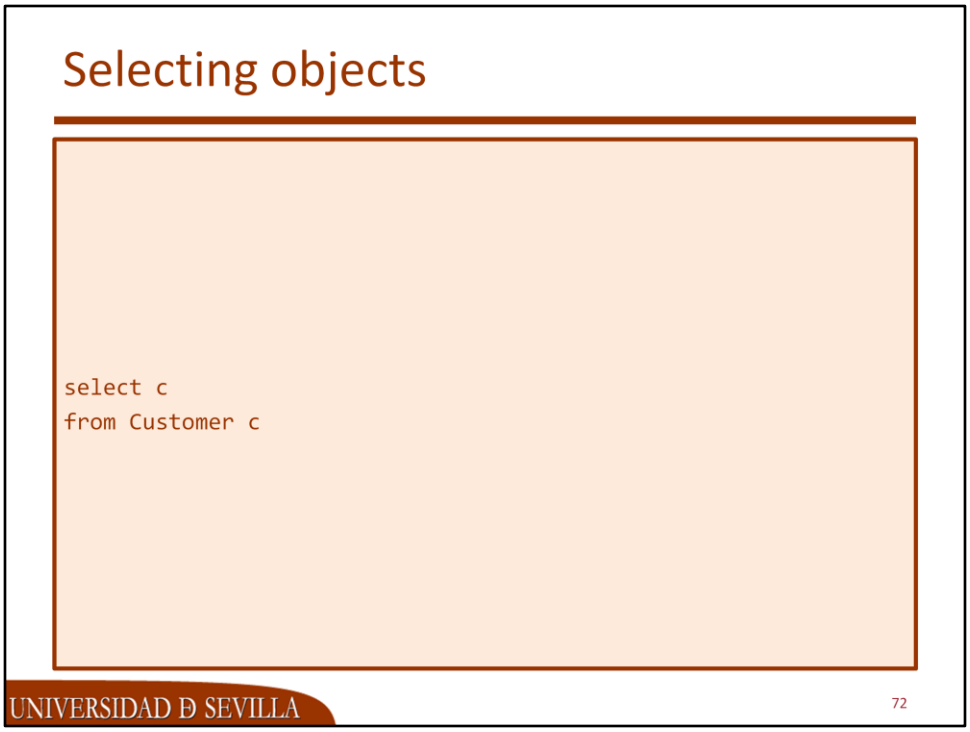



Let's now delve into the details of this section. We'll first explore the queries that allow to select objects from a database; we'll conclude with an insight into the most common operators and functions and a few tips on a utility that we provide so that you can run your queries.

**NOTE:** we won't provide any details regarding the queries to manipulate objects, that is, queries that update or delete objects from the database. The reason is that you're not likely to use them. Typically, you'll update or delete your objects by means of a repository. We deal with such repositories in Lesson "L04 – Repositories and services". Note, too, that we only refer to updating or deleting objects since JPQL doesn't provide any means to insert objects in a database.




Let's start with the queries to select objects.




The JPQL statement to select objects from a database is the select-from-where statement. In its simplest form, it looks like in this slide: “select alias from class alias”. For instance, the example in this slide returns a collection with all of the objects of class “Customer” that were persisted to our database.

## Selecting and filtering

```
select c
from Customer c
where c.id = 7
```



The example in this slide shows a query that selects a the customer whose id is “7”, if any. If such a customer doesn’t exist, it then returns a “null” value.

## Selecting, navigating, and filtering

```
select c
from Customer c
where c.registrations.size > 0
```



This example is a little more complex: this query selects the customers who have registered to at least one announcement. Note that you simply navigate from your customer “c” to his or her registrations and then check whether the size of this collection is greater than zero or not.

## It’s simple!

UNIVERSIDAD Ð SEVILLA

75

For the vast majority of students, it suffices with the previous two examples to get the idea of select statements. You select objects from classes, navigate through their properties, and write constraints on their values. Most of our students will easily find that features such a “distinct”, “group by”, “order by”, or “having” are also available in JPQL and that their syntax is identical to SQL. We should, however, try a few interesting queries before concluding.

## Try it yourself, and learn a new thing!

How would you write a query to select the customers who have registered to the announcement whose id is “7”?

UNIVERSIDAD Ð SEVILLA

76

How would you write a query to select the customers who have registered to the announcement whose id is “7”? Please, try it before you go on to the next slide.

## Your usual first attempt

```
select c
from Customer c
where c.registrations.announcement.id = 7
```

UNIVERSIDAD Ð SEVILLA

77

You might think of something like the statement in this slide, but it doesn't work. The reason is that "c.registrations" refers to a property of type "Collection<Announcement>"; that means that you can query property "size" to find out the size of the collection, but "c.registrations.announcement" doesn't make sense. Type "Collection<Announcement>" doesn't have a property called "announcement".

# The right query

```
select c
from Customer c join c.registrations r
where r.announcement.id = 7
```



To navigate through a property of type collection you need to use a query like the one in this slide; the construct “from Customer c join c.registrations r” creates an alias called “r” by means of which we can navigate individual registrations; “r” iterates over the collection of registrations of every customer, so that “r.announcement.id = 7” works well. It’s important that you understand this query. We’re pretty sure you won’t have any problems with queries that involve one-to-one or may-to-one relationships, but one-to-many or many-to-many relationships are not intuitive in JPQL.

## Try it yourself, and learn a new thing!

How would you write a query to select the names of the customers and the number of registrations they have?

UNIVERSIDAD Đ SEVILLA

79

How would you write a query to select the names of the customers and the number of registrations they have?

## Your usual first attempt

```
select r.owner.name, count(r)
from Registration r;
```



This might well be your first attempt, but it doesn't work. The reason is that this query returns something like "[Customer 1, 5]", where "Customer 1" is the name of a customer and "5" is the total number of registrations. It doesn't return a collection of customer names and the count of registrations to which they've registered.

## The right query

```
select r.owner.name, count(r)
from Registration r
group by r.owner;
```



We need to use a “group by” clause in order to group the registrations by customer and then select their names and count of registrations. This query will return something like “[Customer 1, 2], [Customer 2, 3]”.

## Try it yourself, and learn a new thing!

How would you write a query to select the names and the number of registrations of the customers who have at least three registrations?

UNIVERSIDAD Ð SEVILLA

82

How would you write a query to select the names of the customers and the number of registrations they have, as long as they have at least three registrations?

## Your usual first attempt

```
select r.owner.name, count(r)
from Registration r
where count(r) >= 3
group by r.owner;
```



This is your usual first attempt, but it results in a SQL Exception with message “Invalid use of group function”. You cannot use a statistical function in the where clause.

## The right query

```
select r.owner.name, count(r)
from Registration r
group by r.owner
having count(r) >= 3
```



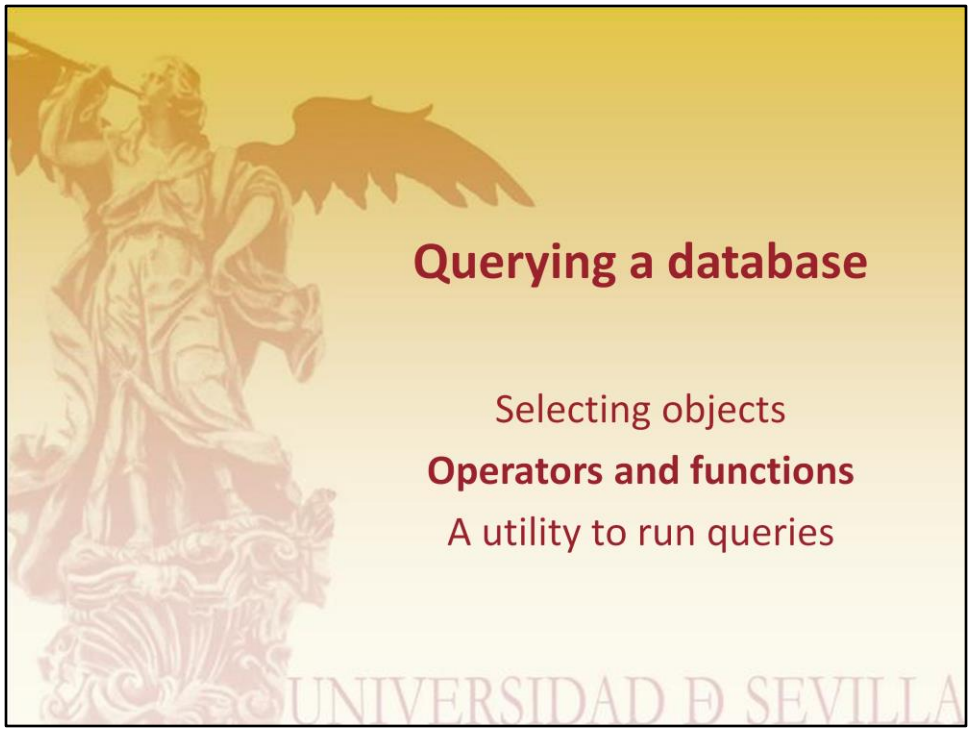
The solution is that your first have to group your results and then filter them using a “having” clause.

## No more details

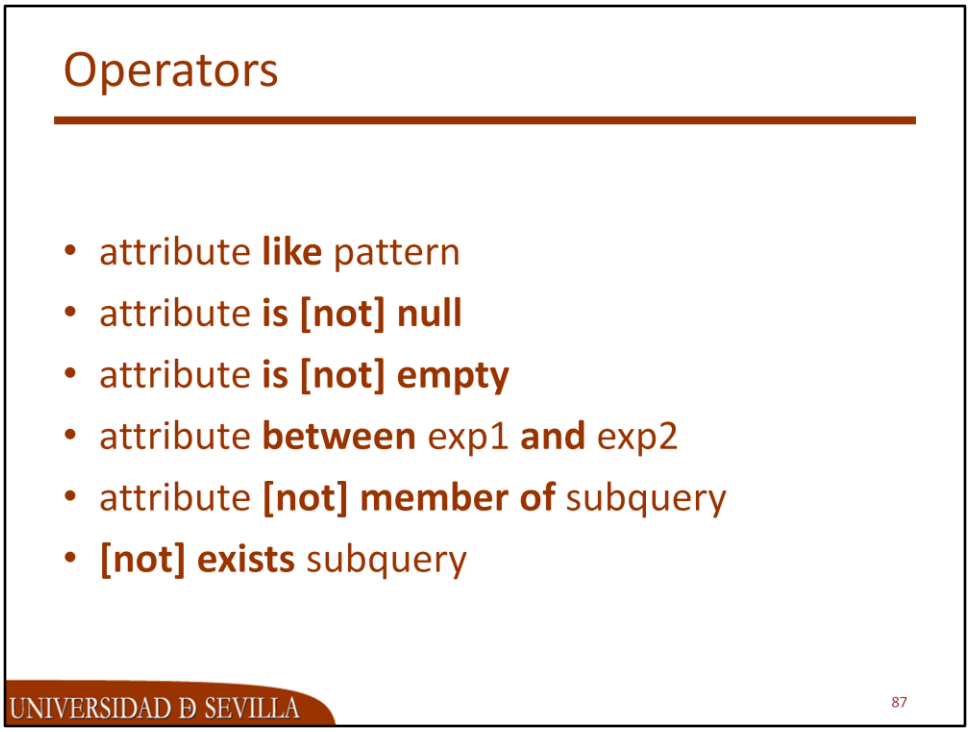

UNIVERSIDAD Ð SEVILLA

85

We won't provide any more details since JPQL is very similar to SQL, which you're expected to command, and we're going to practice a lot. We think that practicing's a lot more interesting that reporting on the boring JPQL syntax and semantics.




Let's now provide a foundation on operators and functions.




This slide reports on the most common JPQL operators:

- “attribute like pattern”: searches for a SQL-like pattern; please, avoid it since it’s very inefficient!
- “attribute is [not] null”: to check if an attribute is (not) null (the same as comparing it explicitly to the “null” value).
- “attribute is [not] empty”: to check if a collection is empty or not (the same as checking if its size is zero or not).
- “attribute between exp1 and exp2”: the same as “attribute >= exp1 and attribute <= exp2”, but more English-like.
- “attribute [not] member of sub-query”: checks if the value of an attribute is (not) in the results of a sub-query.
- “[not] exists sub-query”: checks if the result of a sub-query is empty or not.

## Operator examples

```
select c
from Customer c
where c.registrations is empty
```

```
select q
from Question q
where maximumMark
      between 0.5 and 1.0
```

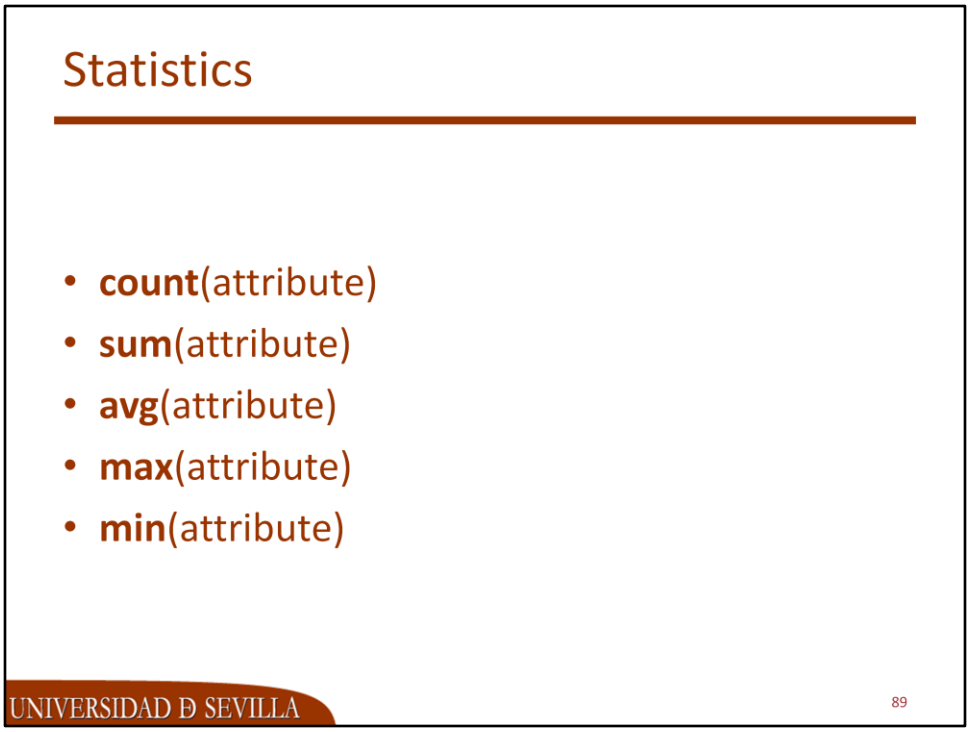
```
select c
from Customer c,
     Registration r
where r member of c.registrations
and   r.announcement.id = 19
```

```
select e
from Exam e
where exists (
       select a
       from Announcement a
       where e = a.exam
)
```



This slide shows a few examples that use operators. From left to right, top to bottom:

- Selects the customers who don't have any registrations.
- Selects the questions whose maximum mark's between 0.5 and 1.0.
- Selects the customers who have registered to an announcement whose identifier's "19".
- Selects the exams that are involved in at least one announcement.




In addition to operators, JPQL also provides a number of useful statistical functions:

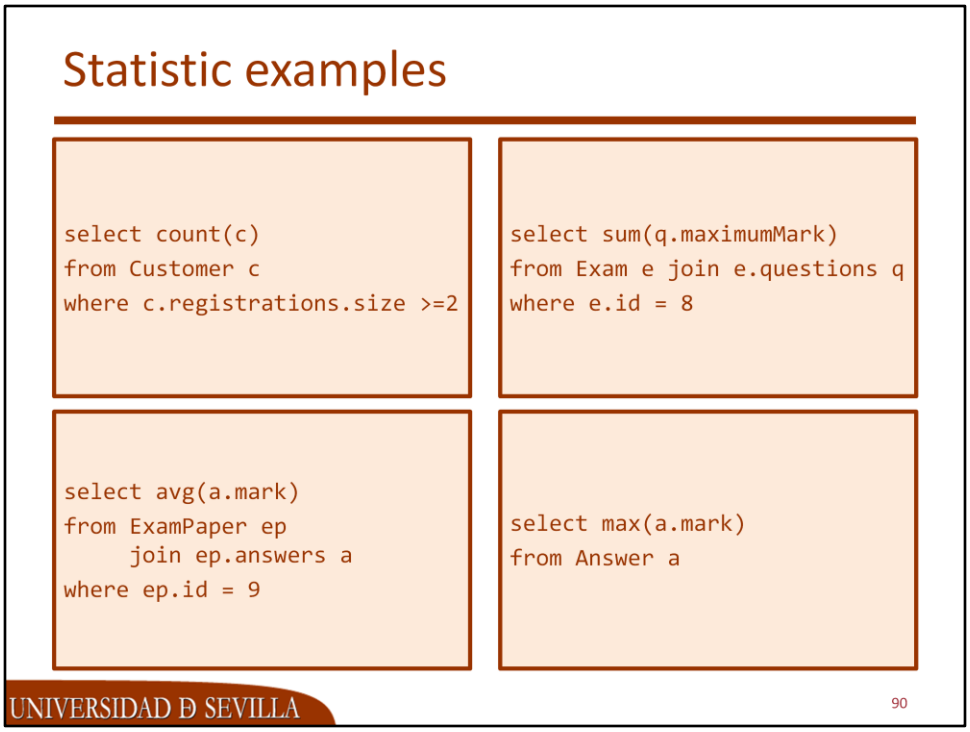
- "count(attribute)": counts the number of instances of an attribute.
- "sum(attribute)": returns the sum of a numeric attribute over the results of a query.
- "avg(attribute)": returns the average of a numeric attribute over the results of a query.
- "max(attribute)": returns the maximum of a numeric attribute over the results of a query.
- "min(attribute)": returns the minimum of a numeric attribute over the results of a query.

Please, note that there's no function to compute the standard deviation of an attribute. If you have to compute such a statistic, then use the following formula, which provides a stimation of the population standard deviation:

```
stdev(X) = sqrt(sum (X * X) / count (X) – avg(X) * avg(X))
```

For instance, if you wish to compute the standard deviation of the fee that customers have to pay for their certifications, then use the following query:

```
select sqrt(sum(c.fee * c.fee) / count(c.fee) - (avg(c.fee) * avg(c.fee)))
from Certification c;
```




These are a few examples that use statistical functions. From left to right, top to bottom:

- Computes the number of customers who have at least two registrations.
- Computes the sum of maximum marks of the questions of the exam whose identifier is “8”, that is, the maximum mark of this exam.
- Computes the average mark of the answers in the exam paper whose identifier “9”.
- Computes the maximum mark of all answers in all exam papers.

## Common functions

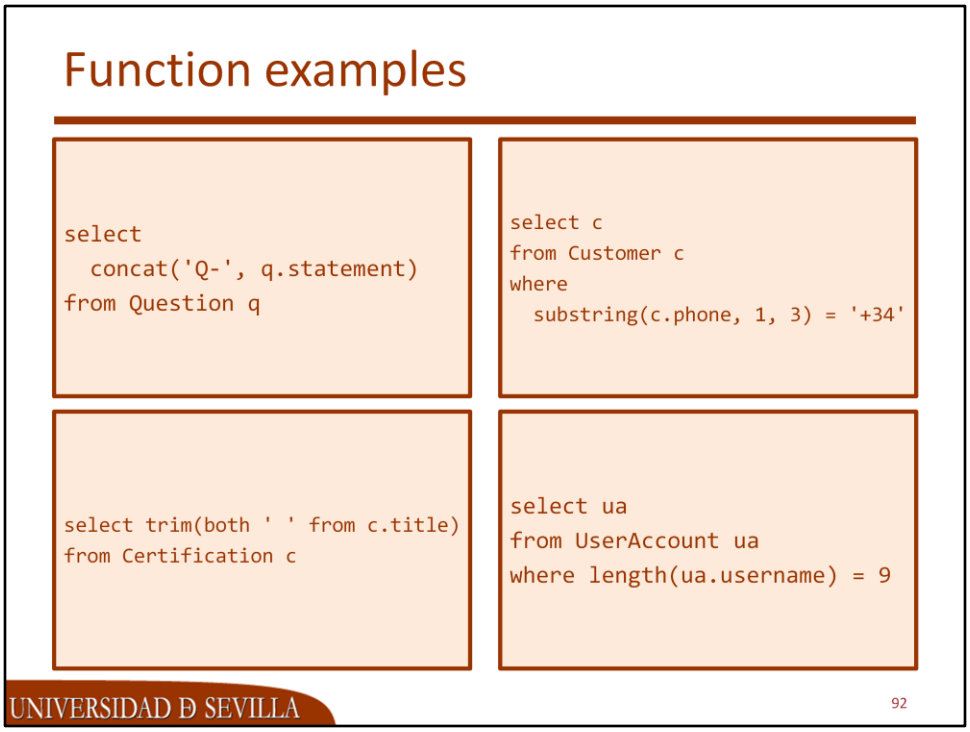
- **concat**(string, string)
- **substring**(string, start-index, length)
- **trim**([**leading**|**trailing**|**both**] char **from** string)
- **length**(string)
- **abs**(number)
- **sqrt**(number)
- **current_timestamp**()

UNIVERSIDAD Ð SEVILLA

91

Finally, we report on some additional common functions:

- "concat(string, string)": concatenates two strings.
- "substring(string, start-index, length)": returns a substring of a string.
- "trim([leading|trailing|both] char from string)": trims a string using the specified character. The syntax of this function isn't usual, so it deserves a little more explanations. For instance "trim(leading 'x' from 'xxxaaaxxx')" returns "aaaxxx"; that is, it trims character "x" from the beginning of the string. Did you get the idea? You specify a character and the function removes it from the beginning, the end, or both ends of a string.
- "length(string)": returns the length of a string.
- "abs(number)": returns the absolute value of a number.
- "sqrt(number)": returns the squared root of a positive number.
- "current_timestamp()": returns the current time date and time.




These are a few examples that use the previous functions. From left to right, top to bottom:

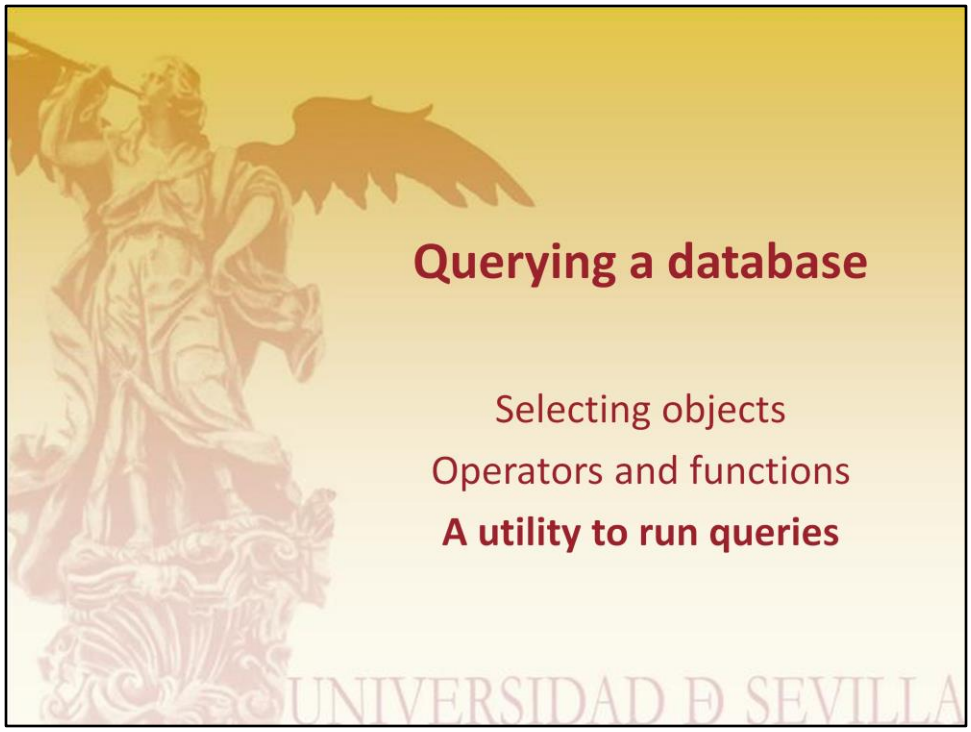
- Returns a collection with the statements of every question in the database prepended with a “Q-”.
- Selects the customers whose telephone number starts with “+34”.
- Returns a collection with the titles of every certification in the database, after trimming their spaces.
- Selects the user accounts whose user name has a length of nine characters.

## There are many others


UNIVERSIDAD Ð SEVILLA

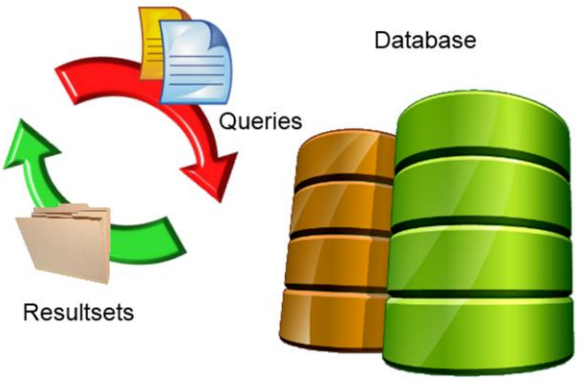
93

We have just scratched the surface. There are many functions available. Please, consult the recommended bibliography if you need additional ones.




Right, we now know quite a lot about JPQL and we're ready to start trying our own queries. Let's provide some details on the utility that we provide.

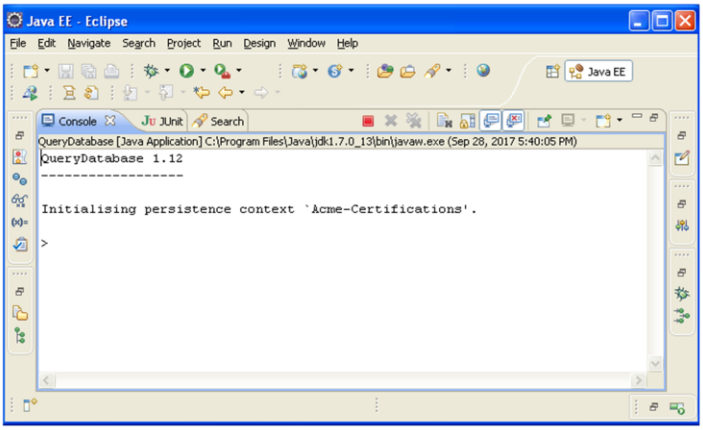
# The QueryDatabase utility






Simply put: the “QueryDatabase” utility is an application that reads queries from the console, runs them through your database, and displays the results on the console.

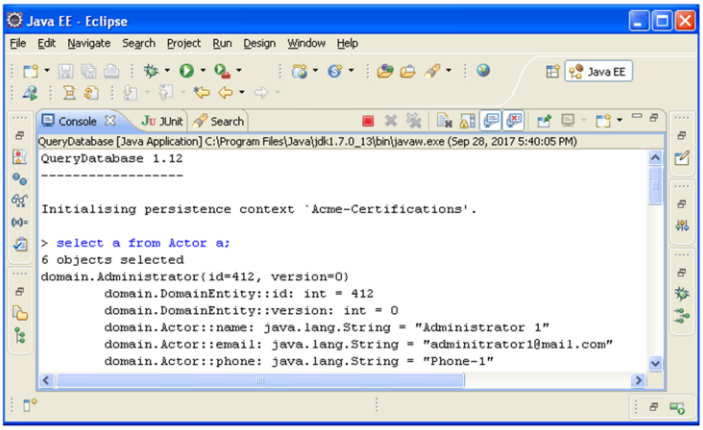
# The console





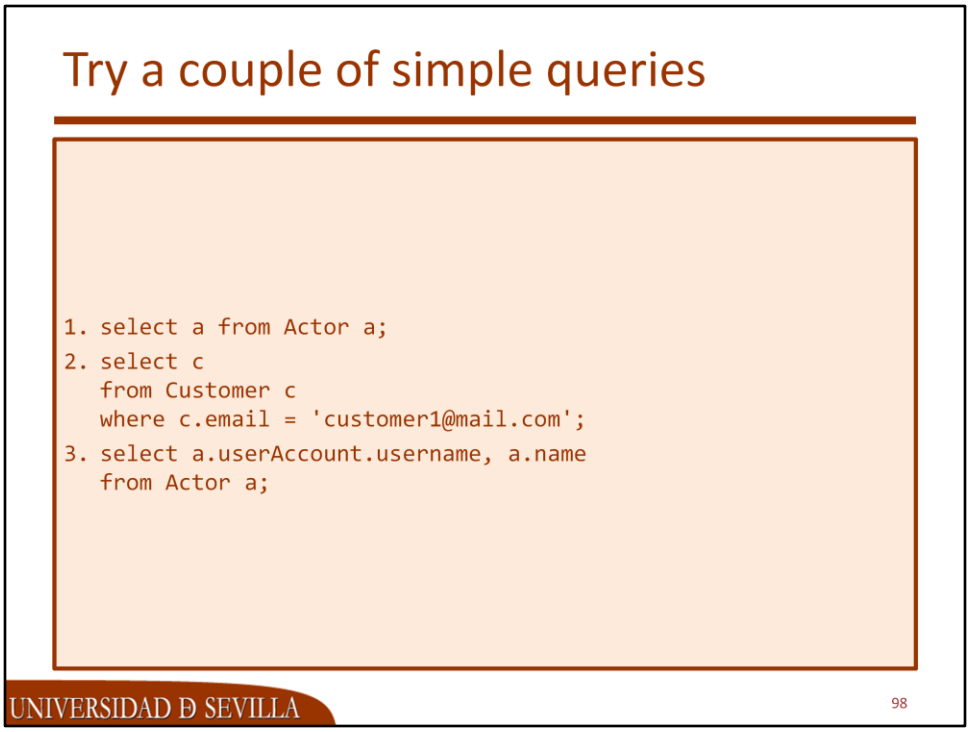

To run it, search for a file called “QueryDatabase.java” in folder “src/main/java/utilities”. Right click it, and select “Run > As Java application”. This is what you should see, a blank console.

# Issuing a simple query



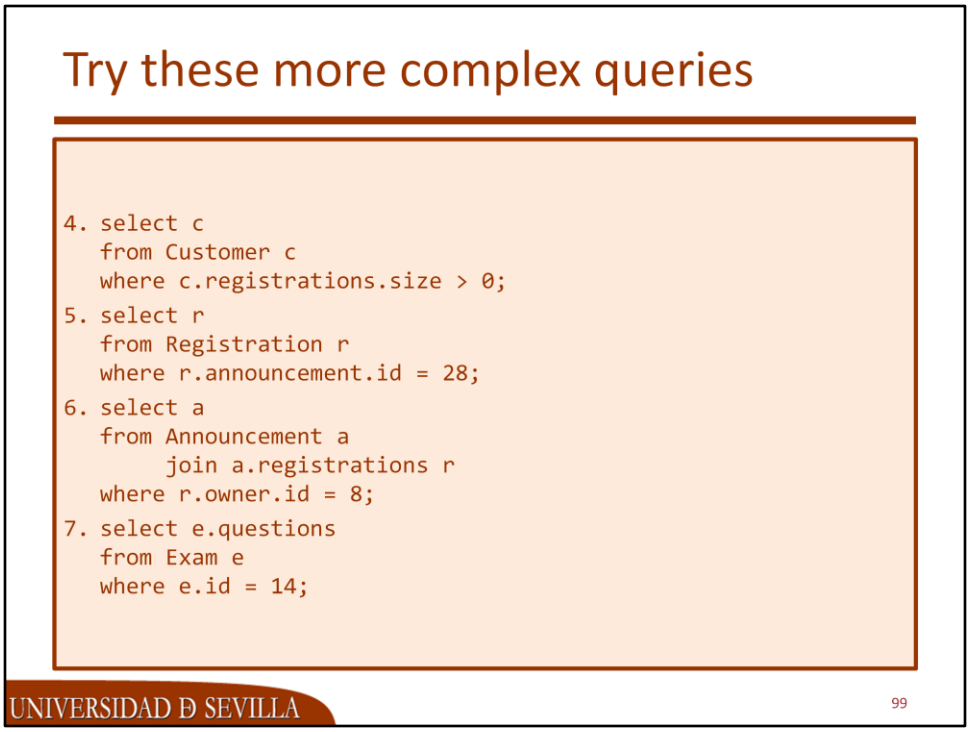



Type the following query in and hit enter: “select a from Actor a”. You’ll get something that’s very similar to the screenshot in this slide. Note that after each query you get a message that indicates the number of objects found, and the listing with the objects themselves, if any.




Ok, it's time to try a few simple queries like the ones in this slide:

1. Selects every actor from the database.
2. Selects the customer whose email is "customer1@mail.com", if any.
3. Selects the usernames and the names of every actor in the database.




We strongly recommend that you should run the following queries when you review this lesson, and that you should try to find their meaning before reading the following notes:

4. Selects the customers who have at least one registration.
5. Selects the registrations for the announcement with identifier “28”.
6. Selects the announcements that have at least a registration for the customer whose identifier is “8”.
7. Selects the questions that compose the exam with identifier “14”.

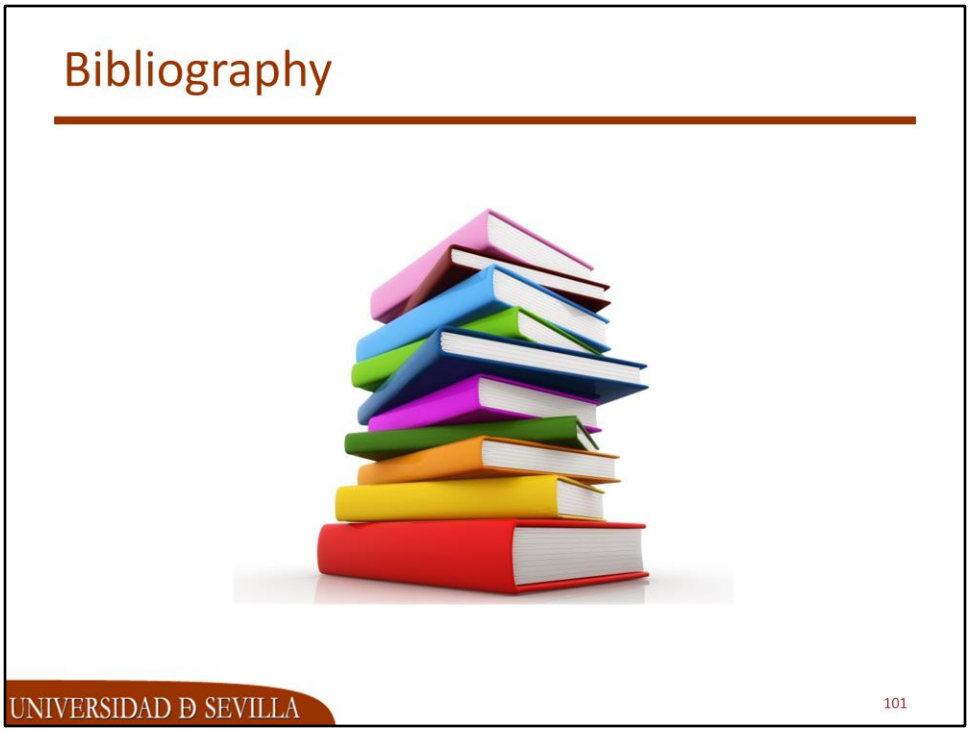
## Try these, too

```
8. select avg(e.questions.size)
   from Exam e;
9. select r, r.announcements
   from Reviewer r;
10.select r, r.announcements.size
   from Reviewer r;
11.select r
   from Registration r
   where r.moment < '2012/01/31';
12.select r
   from Registration r
   where r.moment between '2010/01/01 00:00'
                      and '2012/01/31 23:59';
```



Try the following ones, too:

8. Selects the average number of questions per exam.
9. Selects the reviewers and the announcements to which they are assigned.
10. Selects the reviewers and the number of assignments to which they are assigned.
11. Selects the registrations that where created before January 31, 2012.
12. Selects the registrations that where created between January 1, 2010 at 00:00 and January 31, 2012 at 23:59.




Please, take a look at the following web pages for further information:

- http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/manual.html
- http://www.objectdb.com/java/jpa/

The technology that we use to interpret the "PopulateDatabase.xml" configuration file is called Spring IoC. You may find additional information on this technology at https://docs.spring.io/spring/docs/4.0.0.RELEASE/spring-framework-reference/htmlsingle/#beans.

There's also an interesting book that focuses exclusively on JPA/JPQL:

Pro JPA 2: Mastering the Java Persistence API
Mike Keith, Merrick Schnicariol
Apress, 2010

You might also be interested in the following book:

Oracle database 11g & MySQL 5.6 developer handbook
Michael McLaughlin
McGraw-Hill, 2012

This bibliography is available in electronic format for our students at the USE's virtual library. If you don't know how to have access to the USE's virtual library, please, ask our librarians.

## Time for questions, please




Time for questions, please. Note that we won't update the slides with the questions that are posed in the lectures. Please, attend them and take notes.

# The next lecture


- We need (coerced) volunteers
- Volunteer collaboration is strongly advised
- Produce a solution and a presentation
- Rehearse your presentation at home!
- Each presentation is allocated 100/*N* min
- Presentations must account for feedback



The next lecture is a problem lecture. We need some volunteers, who are expected to collaborate to produce a solution and a presentation. Please, rehearse your presentation at home taking into account that you have up to 100/*N* minutes per problem, including feedback, where *N* denotes the number of problems.


Thanks for attending this lesson!