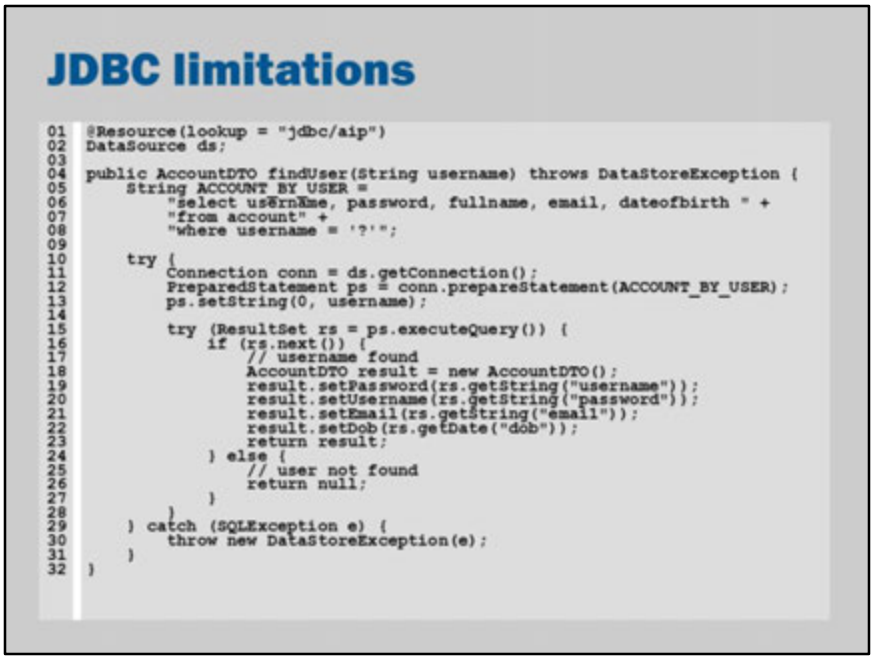
Object-relational



Can you see the errors in the above code?

1. "dateofbirth" on line 06 vs "dob" on line 22
2. "email" on line 06 vs "emai1" on line 22 (last character is the number 1)
3. "fullname" is retrieved on line 06 but not used
4. Space is missing after "from account" on line 7
5. Quotes are around the "?" in the prepared statement on line 8. It should just be "where username = ?"
6. The connection is created on line 11 but not closed
7. The prepared statement on line 12 should also be closed, as a matter of good practice
8. ps.setString(0,…) instead of ps.setString(1,…) on line 13 (JDBC starts numbering from 1)
9. Username is set to password on line 19
10. Password is set to username on line 20

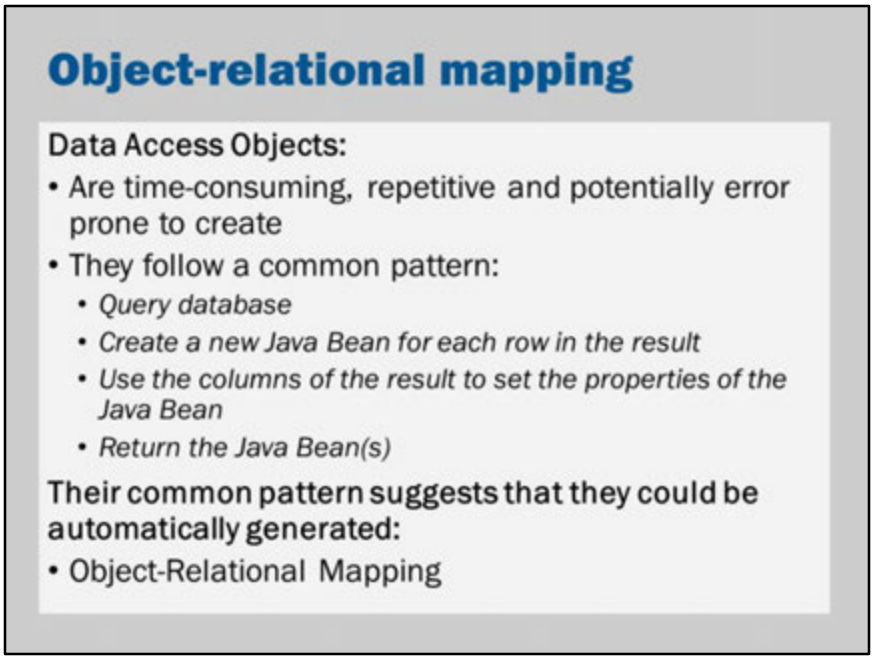
NONE of these errors will be detected by the compiler!

Some of the errors would not even cause immediate errors at runtime (e.g., not closing connections works fine for a while until the container runs out of connections

in the pool; not setting the fullname in the DTO might cause information to be missing in a JSF view).

Clearly, JDBC has some problems. On first glance, the code above looks fairly reasonable. It compiles and it looks ok.

It would be better if these problems could be found by the compiler so that you're less likely to get a surprise when you run the application.

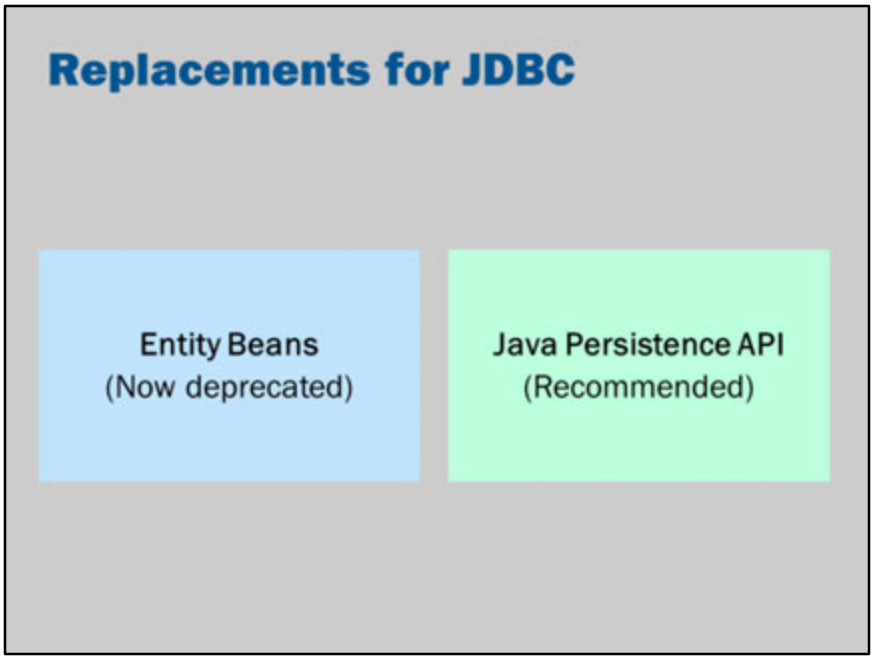


In Assignment 1, you probably created several Data Access Objects.
You would have created a DAO for accounts and another DAO to track the main data of your application.
The code for each DAO would have been fairly similar.
Even in a single DAO, each method follows roughly the same pattern.

As a programmer and/or a computer scientist, repetition is a sign that something could be improved.

Object-Relational Mapping (ORM) is the name of a technology that recognizes that storing and retrieving data from a database follows a common pattern.

ORM attempts to automate the process of mapping to/from database rows to objects.

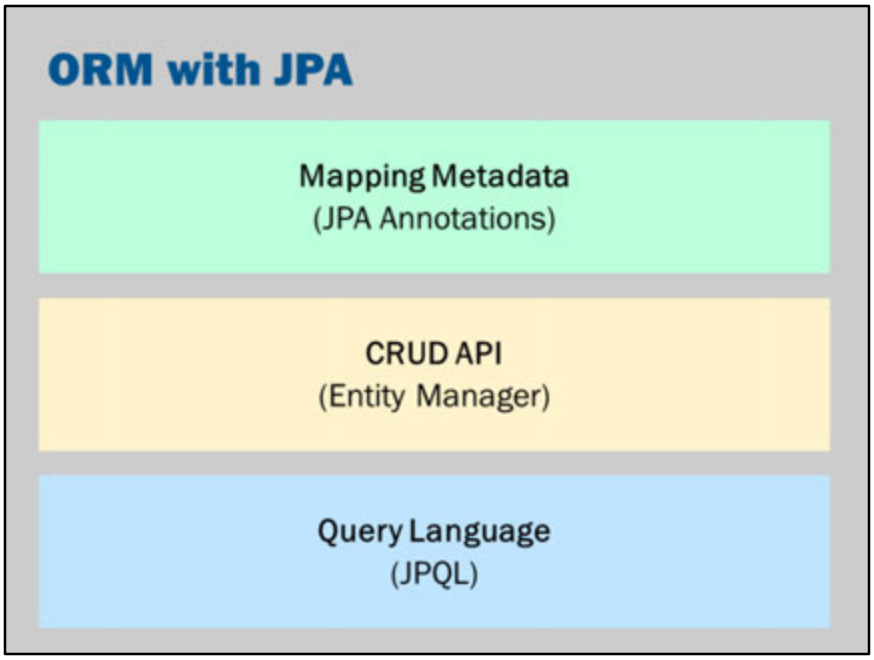


In Java EE there are two main alternatives to using JDBC.

Entity Bean
An Entity Beans is an Enterprise Java Beans (EJB).
An Entity Bean is similar to a row in a database.
It is identified by a primary key and can be persisted to a database.
It is no longer a mandatory part of the Java EE specifications and is recommended for removal.
The problem with Entity Beans is that every EJB uses many resources.
EJBs are server-side components.
When a remote client accesses an EJB, the object is NOT sent over the network.
Instead, the client establishes a network connection and then every method call on the EJB is a separate network call.
Thus, if you have many hundreds of Entity Beans, you need to manage many hundreds of remote network connections.

Java Persistence API (JPA)
JPA is now the recommended persistence technology.
In JPA, Entities correspond (roughly) to rows in a database.

However, Entities are not EJBS. Entities are POJOs (Plain Old Java Objects).
This means that they do not use many resources.
This also means that the whole object can be sent over a network to a remote client: there is no need to retain a server-side component.
JPA Entities are like the DTOs used in a DAO (and also elsewhere where the DTO pattern is used).

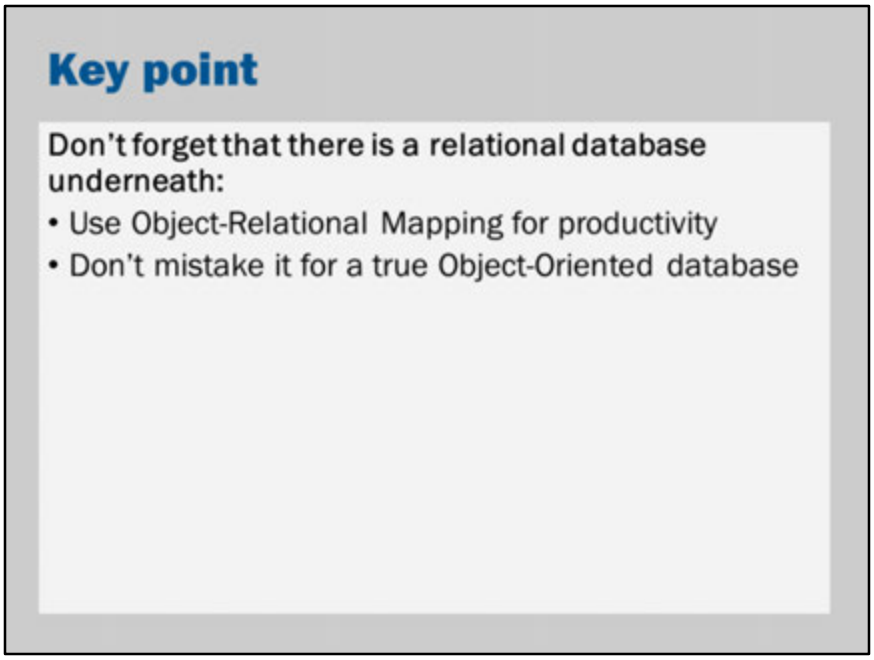


In JPA, there are three main concepts:

Entity Manager
EntityManager is the "general purpose" Data Access Object in JPA.
Create, read, update, delete operations are performed using the Entity Manager.

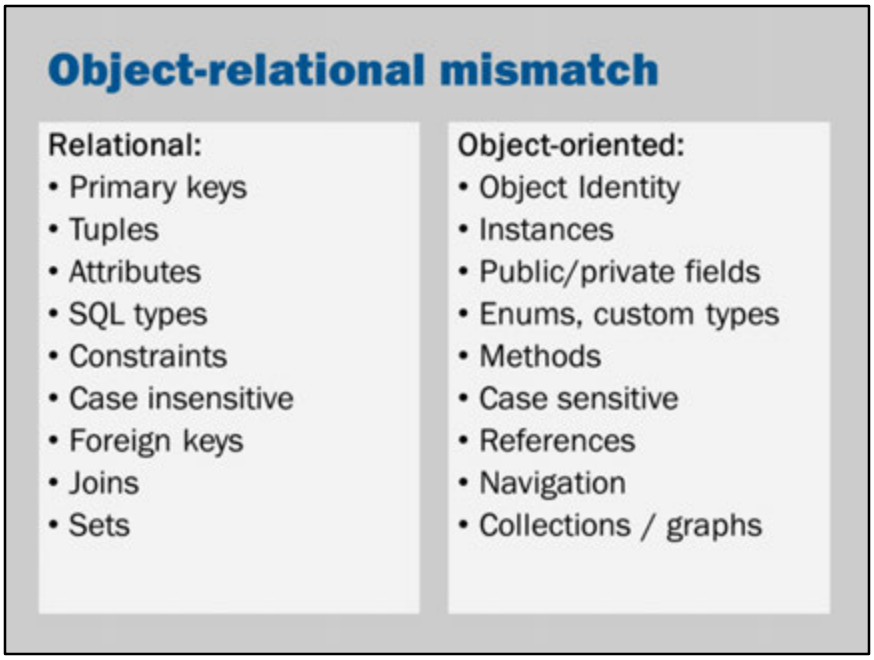
Query Language
Java Persistence Query Language (JPQL) is the query language used by JPA.
If you know SQL, then JPQL should not be difficult.
You use the EntityManager to execute JPQL queries.

Mapping Metadata
JPA Annotations are used to tell JPA how to translate between Java classes (entities) and database tables/columns.
JPA has a default, automatic mapping between Java names and database names.
However, you can also use annotations to override the default behavior.



Object-relational mapping is very powerful and does a good job of hiding the details of the underlying database.
However, the abstraction isn't perfect.
The object-relational mismatch is important.
Object-relational mapping cannot hide all the philosophical differences between relational databases and object oriented models.
Ignore these differences at your own peril!
If you forget that your Entities are a mapping of a database, then you will run into problems.
It is better to think of JPA as a "very powerful" way of creating SQL queries, rather than expecting that JPA provides a perfect object-oriented database.

Entity annotations



There are some fundamental philosophical differences between the object oriented model and relational databases.

A relational database is based on mathematical set theory.
Each record is a database is just a "value".
Two records with the same value are the same.

In contrast, the object oriented model is based around the concept of objects that have unique identity.
It is possible to have two objects that have the same values but different identity (and so they are not the "same").

Other differences are listed in the table above.

Yet more points of distinction between the relational model and object oriented models are:

- Focus on data *versus* focus on 'objects' or 'nouns'
- Fixed size string types (VARCHAR) *versus* unbounded Strings
- Data sorted by indexes *versus* data sorted by sorting functions

- No concept of encapsulation or private data *versus* encapsulation and private fields
- Atomic attribute values *versus* structs and other compound types
- Referential integrity constraints and cascading deletes *versus* references and garbage collection
- Manipulation via SQL commands *versus* direct manipulation of fields

And more:
- Blobs
- Streaming
- Transactions
- Uniqueness constraints

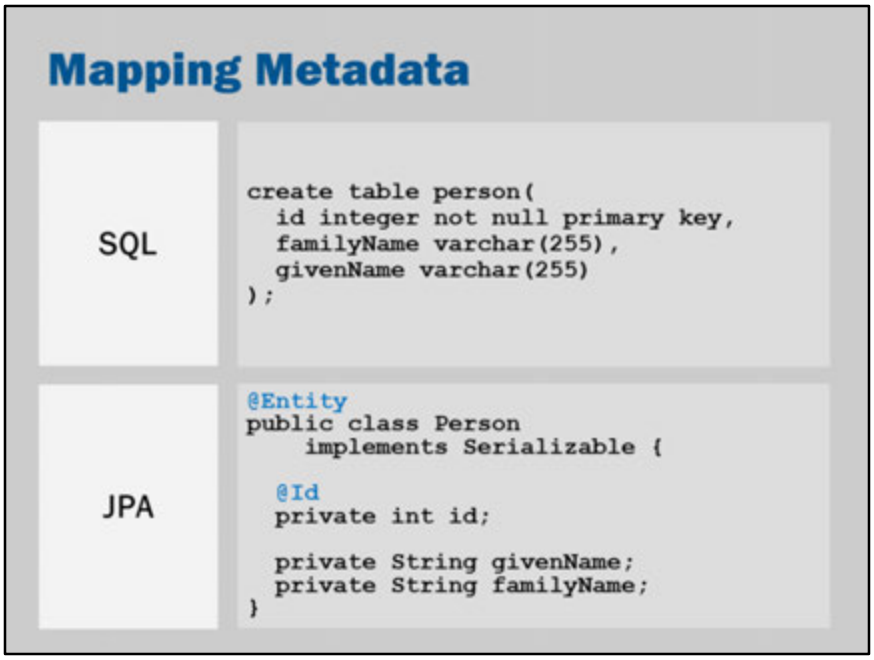
Considering all these differences, it is remarkable that object-relational mapping (ORM) is at all possible!

If you keep these differences in mind, you will better understand the limitations of ORM (or JPA) and also why JPA has been designed the way it has been designed.

Conceptual vs database design

Object oriented design may differ from database design for a range of reasons:

- Performance
- Compatibility with other systems
- Data 'normalization'



The first questions faced by object-relational mapping might be, "which classes should be mapped to the database?" and "what should the primary key be?"

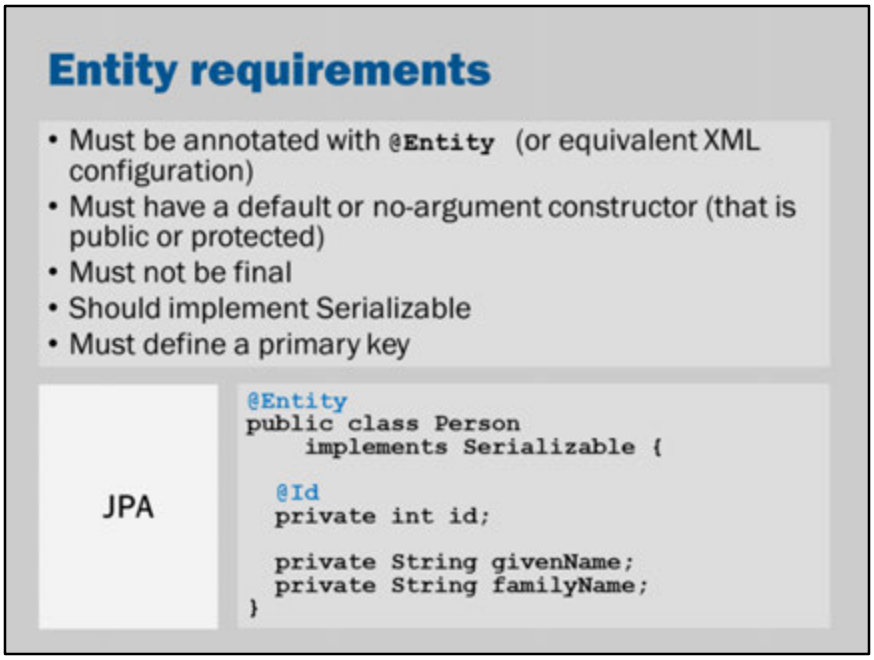
Obviously, it would not be a good idea to map every single Java class to an equivalent database table.

JPA uses the @Entity annotation to mark those classes that should participate in object relational mapping.

The key requirements of an entity are:
It must be annotated with @Entity (or there must be an equivalent configuration in the XML descriptor).
It must have a public or protected no-argument constructor (if you declare no constructors, then java automatically creates a no-argument constructor for you).
It must not be a final class, nor may any persistent variables be final.
It should implement the Serializable interface.
It must have a primary key (e.g., @Id).

The example in this slide shows a database definition and its corresponding Java

entity equivalent.



These are the requirements from section 2.1 of the JPA specifications:

The entity class must be annotated with the Entity annotation or denoted in the XML descriptor as an
entity.

The entity class must have a no-arg constructor. The entity class may have other constructors as well. The no-arg constructor must be public or protected.

The entity class must be a top-level class. An enum or interface must not be designated as an entity.

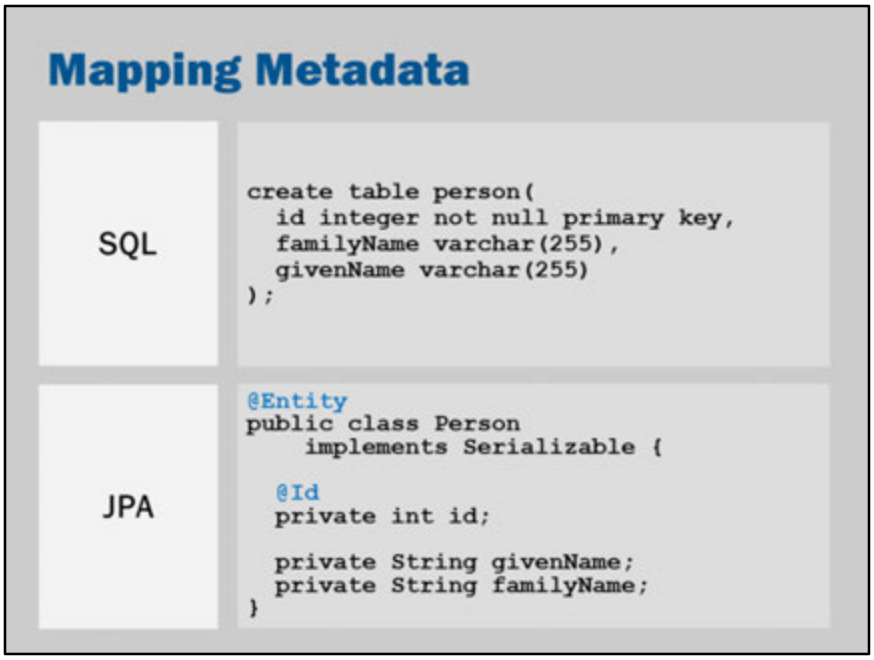
The entity class must not be final. No methods or persistent instance variables of the entity class may be
final.

If an entity instance is to be passed by value as a detached object (e.g., through a remote interface), the
entity class must implement the Serializable interface.

Entities support inheritance, polymorphic associations, and polymorphic queries.

Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as
entity classes, and non-entity classes may extend entity classes.

The persistent state of an entity is represented by instance variables, which may correspond to Java-
Beans properties. An instance variable must be directly accessed only from within the methods of the
entity by the entity instance itself. Instance variables must not be accessed by clients of the entity. The
state of the entity is available to clients only through the entity's methods—i.e., accessor methods (getter/
setter methods) or other business methods.



The first questions faced by object-relational mapping might be, "which classes should be mapped to the database?" and "what should the primary key be?"

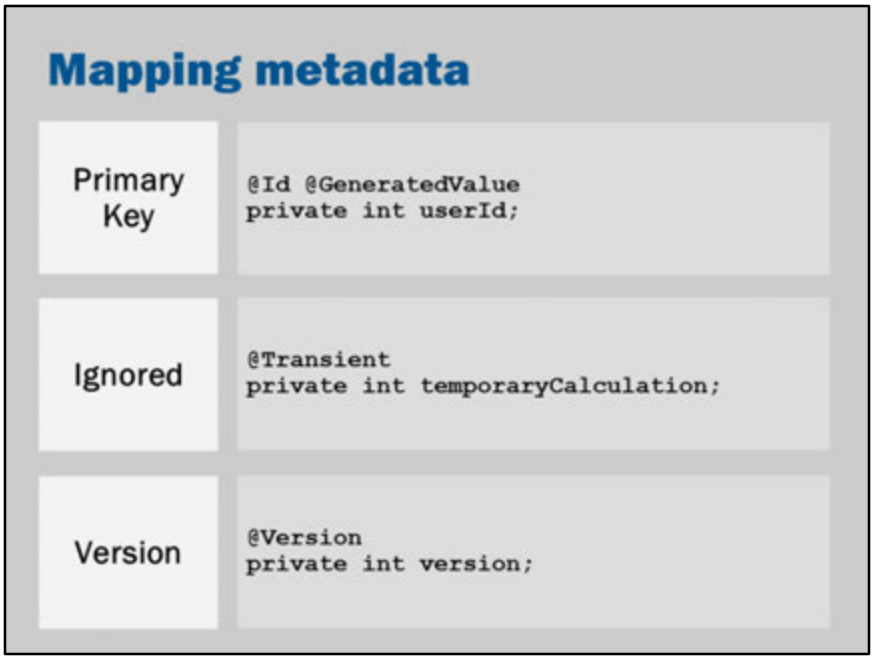
Obviously, it would not be a good idea to map every single Java class to an equivalent database table.

JPA uses the @Entity annotation to mark those classes that should participate in object relational mapping.

The key requirements of an entity are:
It must be annotated with @Entity (or there must be an equivalent configuration in the XML descriptor).
It must have a public or protected no-argument constructor (if you declare no constructors, then java automatically creates a no-argument constructor for you).
It must not be a final class, nor may any persistent variables be final.
It should implement the Serializable interface.
It must have a primary key (e.g., @Id).

The example in this slide shows a database definition and its corresponding Java

entity equivalent.



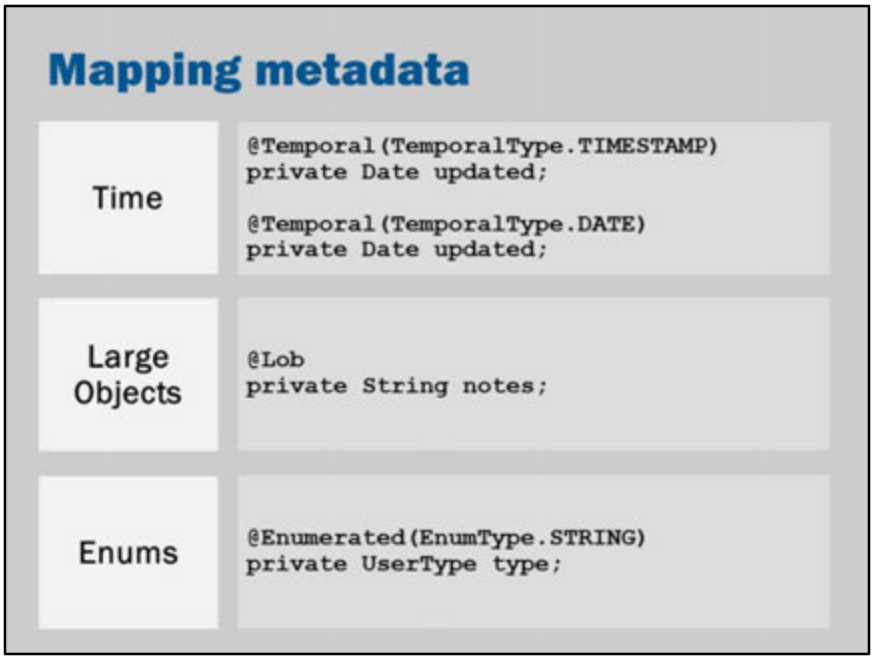
If you want to set the primary key manually, you would declare it like this:
@Id
private int userId;
If you want JPA to automatically generate ids for entities, then you declare it like this:
@Id @GeneratedValue
private int userId;

When JPA sees @GeneratedValue, it will automatically create ids for new objects when they are saved in the database.
The value will be unique (however, it may not increase in steps of exactly one).

@Transient tells JPA to ignore the attribute of the entity. It will not be saved to the database.

@Version is an advanced feature used by JPA to perform optimistic locking.
If you have an @Version attribute, the database will have an additional column to keep track of the entity version.
The version will increase by one every time you make a change to the record in the

database.

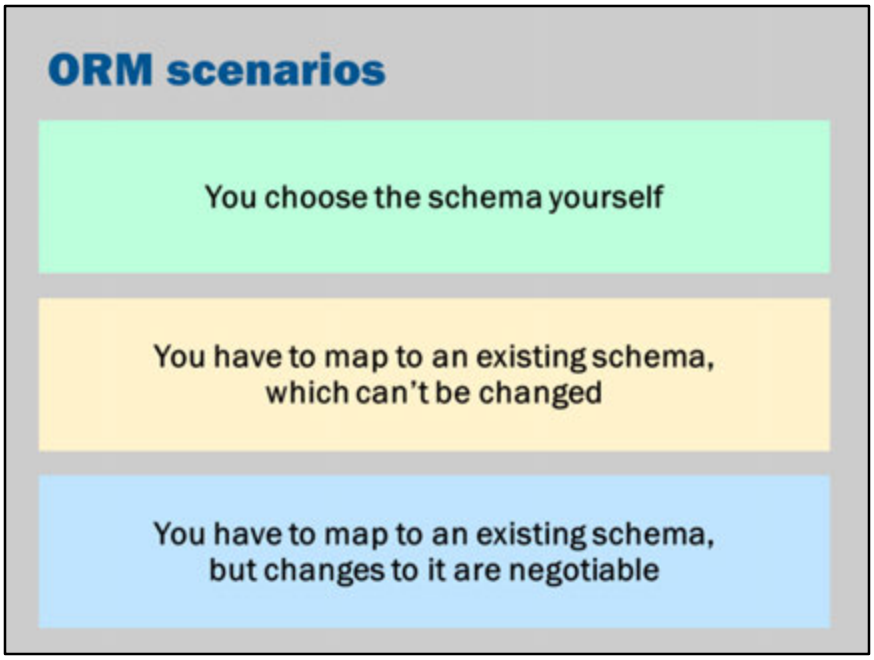


Java data types do not exactly match SQL data types.

The Java Date class may be mapped to SQL Date, Time or Timestamp (combined date and time) values.
To tell JPA which one to use, the @Temporal annotation is used.

Long strings and binary arrays could be stored in VARCHAR.
However, VARCHAR usually has maximum size limits.
Many databases support "Large Object" (LOB), "Character Large Object" (CLOB) or "Binary Large Object" (BLOB) types.
These LOBs usually allow much larger maximum sizes (but also have more overheads).
The @Lob annotation tells JPA to use a LOB type.

Most databases do not have a concept of an enum type.
@Enumerated can be used to tell Java whether to map the enum to a String column (and store by name) or a numeric column (to store by a numeric value).



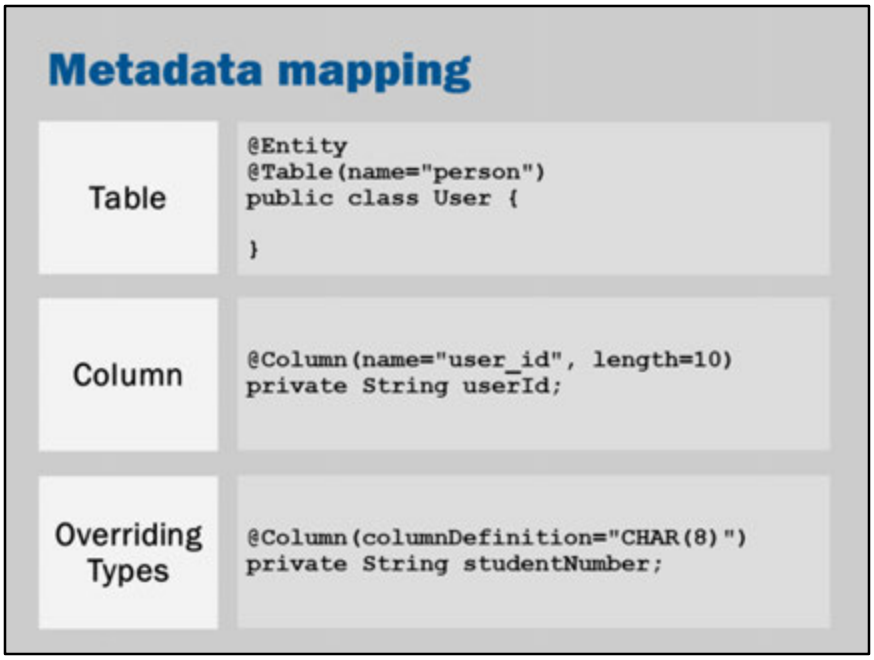
--Fowler, M. (2003) Patterns of Enterprise Application Architecture, Addison Wesley, p. 47.

JPA is designed to support these three scenarios.

Things are easiest if you don't have a fixed database schema.
You can just use the default mappings provided by JPA.
You can let JPA create the database for you.

Things are hardest if you have been given a fixed database schema.
This might be the case if you are working in a large organization that has many legacy systems.
Your new system may be required to work with an existing database.
JPA metadata annotations allow you to precisely control the mapping between Java and SQL.
However, getting the mapping "just right" can be tricky.
Furthermore, you might need to declare additional entities purely for the mapping, that you wouldn't ordinarily create from an object-oriented perspective.

In the middle ground, you might have a scenario where there is already a schema, but the database administrators are happy to change/evolve it.
In this case, you are likely to use JPA annotations to minimize the amount of changes required.
However, some complex relationships are hard to define using JPA.
In this case, you might ask small changes to be made in order to simplify the complexity of dealing with the relationships.



JPA provides annotations that allow you to override the default mapping rules.

You can configure the name of the SQL table, the column names, sizes and types.

"@Entity public class User" might map to "create table user".
You can override it with @Table(name="app_user") and then it would map to "create table app_user".

private String userId, by default, may map to "userid varchar(255)" but this could be overridden with @Column to something like "user_id varchar(10)".

The database types may also be explicitly declared.

Field vs property

Field

```
@Column("full_name")
private String name;

public String getName() {
  return name;
}

public void setName() {
  this.name = name;
}
```

Property

```
private String name;

@Column("full_name")
public String getName() {
  return name;
}

public void setName() {
  this.name = name;
}
```

In the examples so far, the annotations have been on the class fields.
In other words, the instance variables of the class have been annotated.

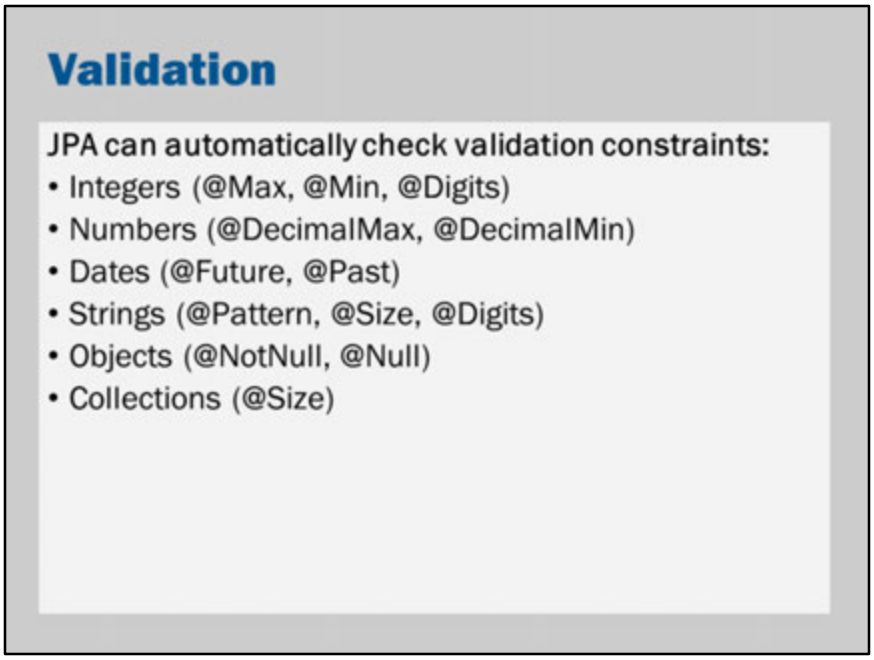
JPA can work with fields or properties.

Properties are the pairs of get/set methods.
If you annotate the get/set methods of a class, then JPA will call your getters and setters to do persistence (rather than saving the instance variables).

You can use one or the other, but not both!

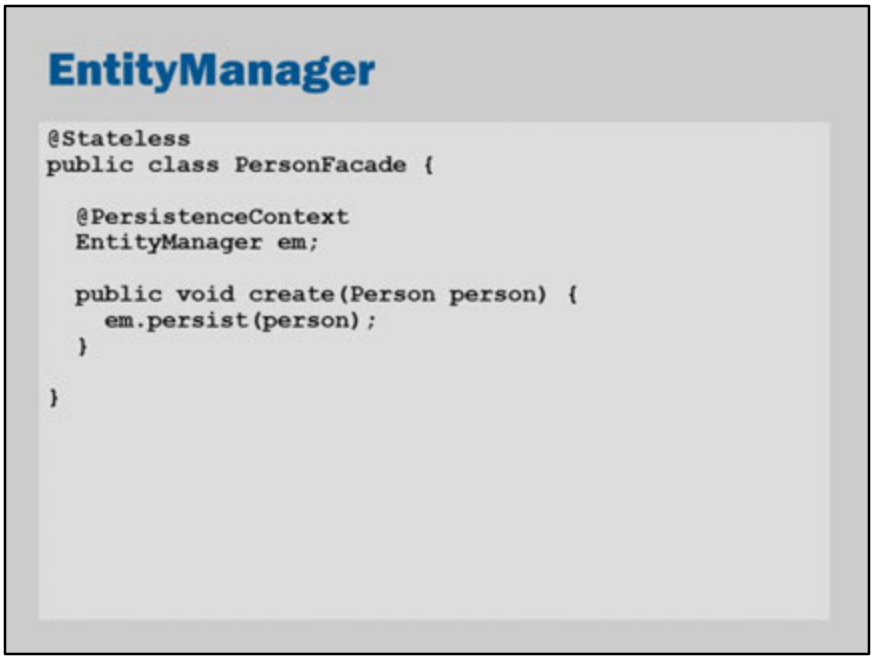
I recommend property-based access because it works more closely with the Java Beans model and inheritance.

That is, an advantage of property-based access is that you can carefully control what happens when data is saved and restored.
You can maintain a complex internal state, but have only a simplified state saved to/from the database.



You can use the same validation attributes that we used earlier with JSF.


Entity manager



An EntityManager is used to create, read, update and delete values from the database.

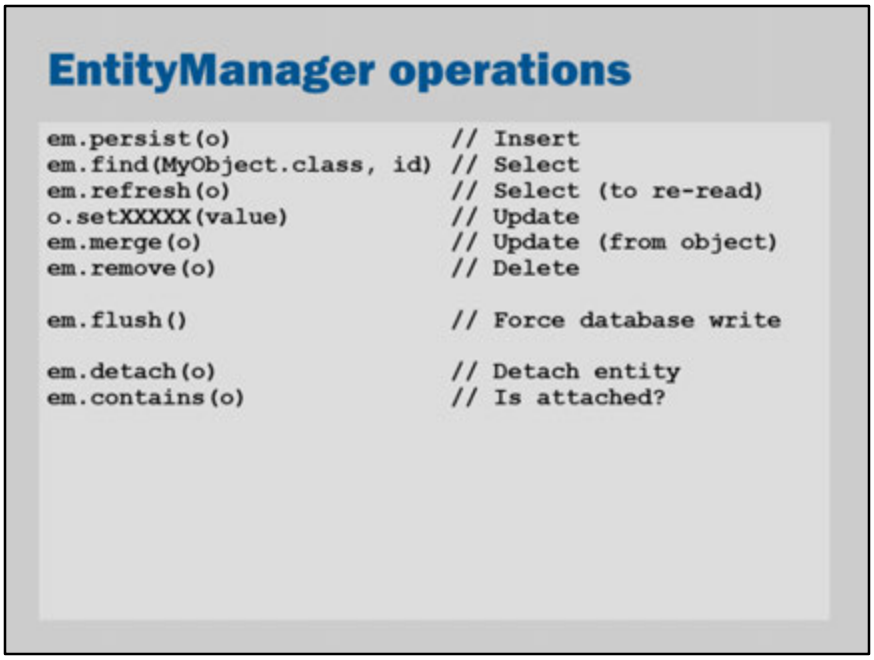
Let's look at the class above:

@Stateless
The class is a Stateless Session Bean. This is because JPA needs to run in a transaction. The easiest way to ensure there is a transaction is to use JPA from within an EJB. In Java EE applications, it makes sense that EntityManager should only be used from EJBs since EJBs are used for implementing the domain logic.

@PersistenceContext
This annotation is used for dependency injection of the EntityManager.
In a sense, it introduces a "persistence context" that the EntityManager will look after.
A persistence context is a collection of entities (objects) held in memory.
It mirrors the contents of the database.
Eventually the persistence context is closed or flushed to the database.
When this happens, all those entities (objects) held in memory are saved to the

database using SQL insert or SQL update statements.

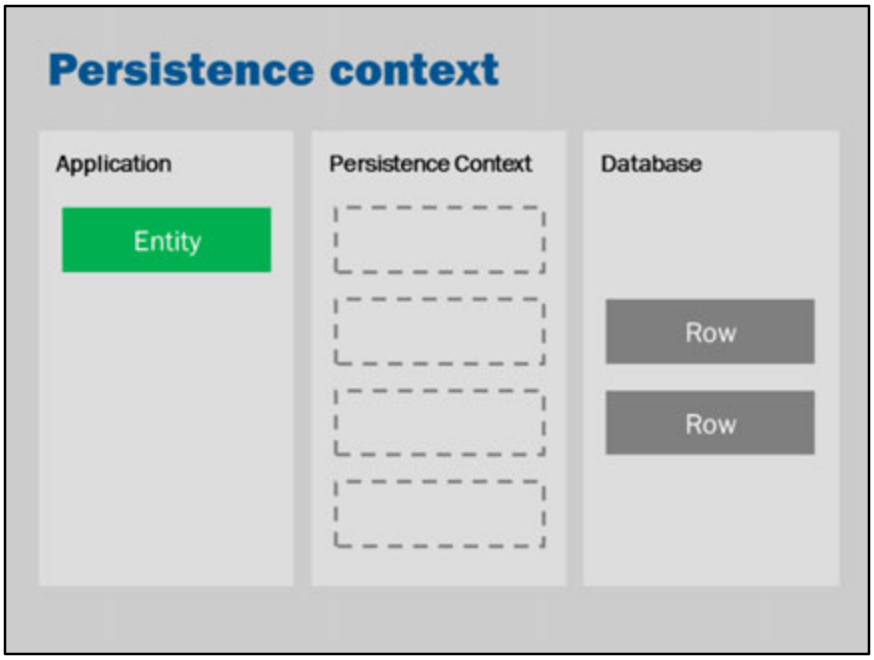
em.persist(person);
This is the "create" operation on an Entity Manager.
em.persist adds the person object to the persistence context.
Conceptually, you might also say that the instance is saved to the database.
However, this does not actually happen until the transaction is closed (i.e., the session bean method returns).
When the transaction closes, the persistence context is saved to the database.



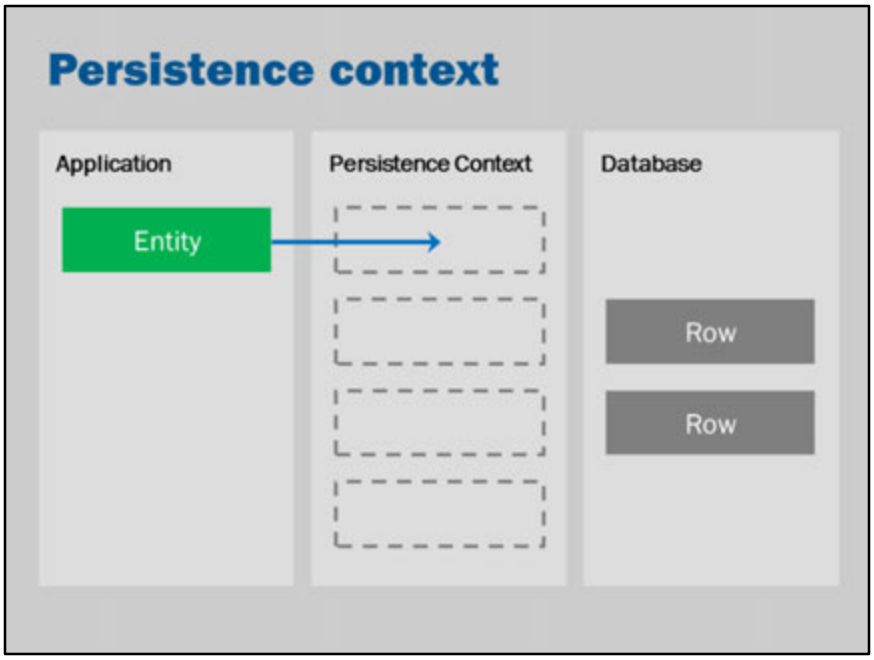
The EntityManager offers create, read, update and delete operations.

The EntityManager works with a persistence context.
Changes don't get saved to the database until the transaction concludes or the changes are otherwise flushed to the database.
The em.flush() forces an immediate save to the database.

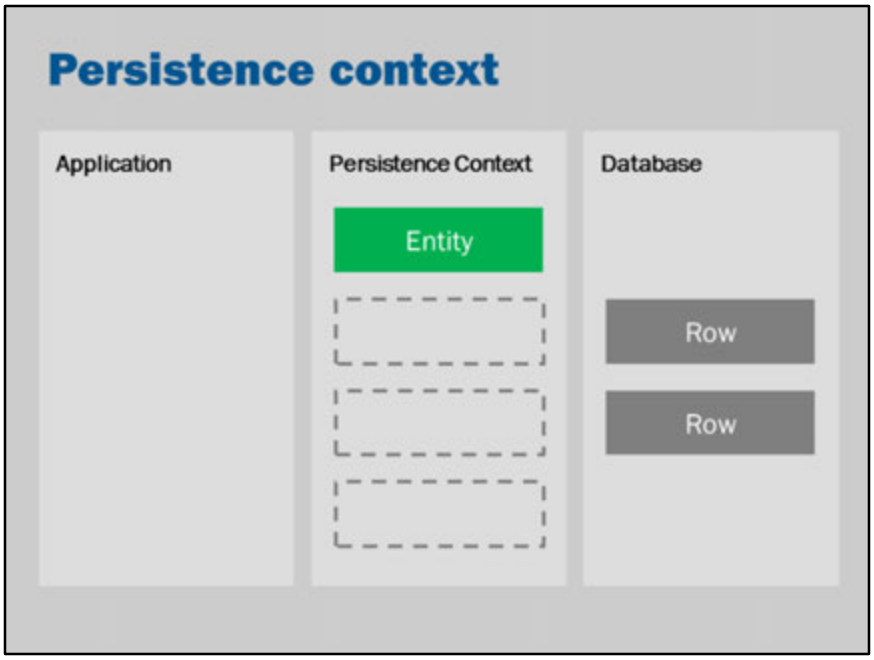
The last two methods, *detach* and *contains*, are used to remove an entity from the persistence context or check if the persistence context currently contains the object.



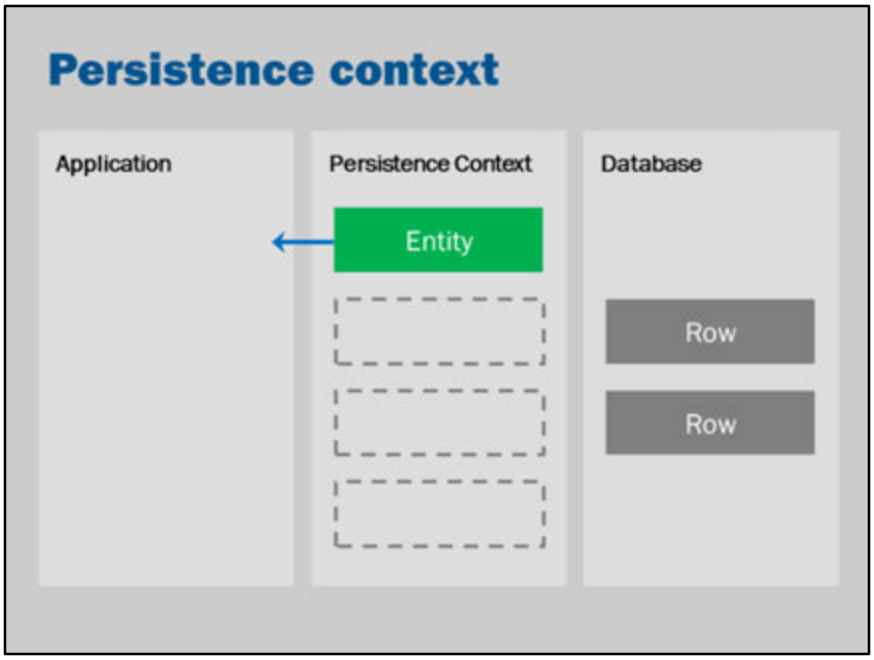
The persistence-context works like an in-memory list.



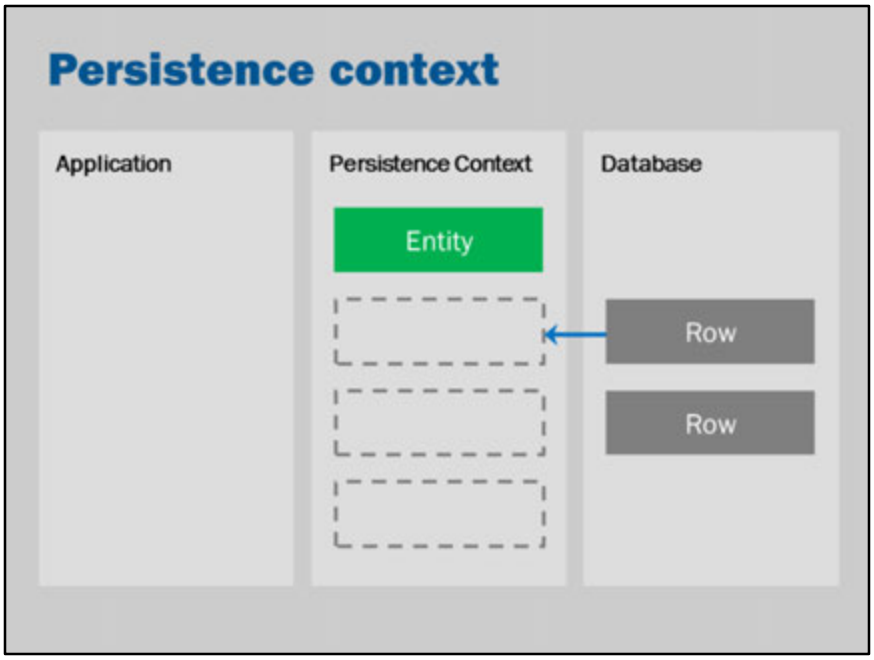
Calling persist, adds to that in-memory list.

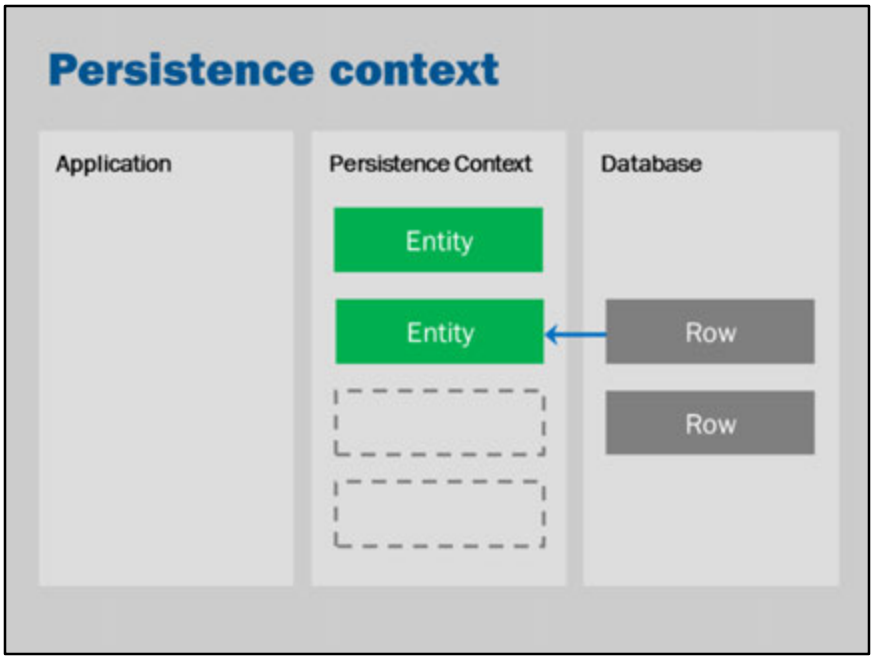


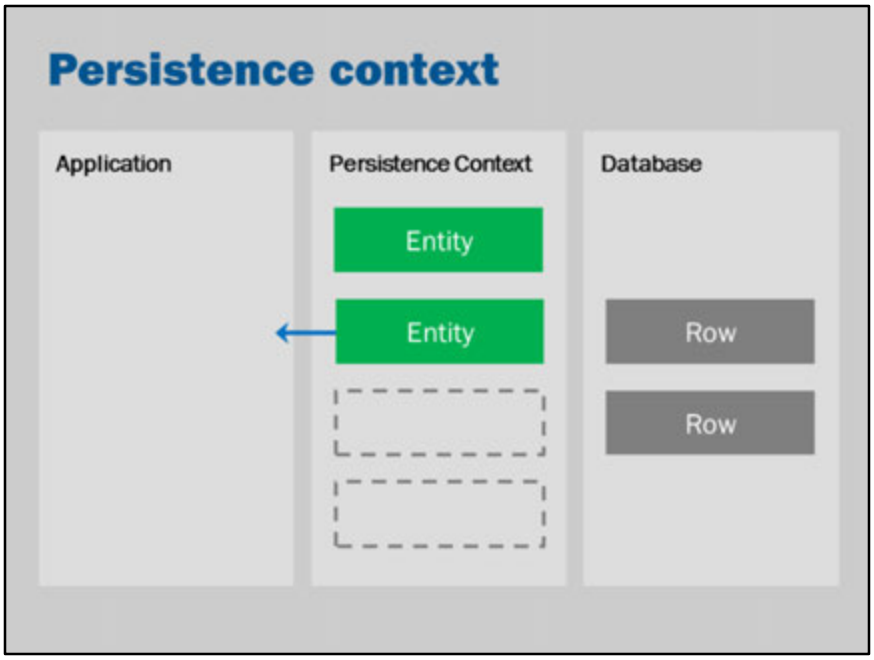
If you call find, but the object isn't in the in-memory list, then JPA will look in the database.
If it finds it in the database, it will load it into memory and then return that to the user.

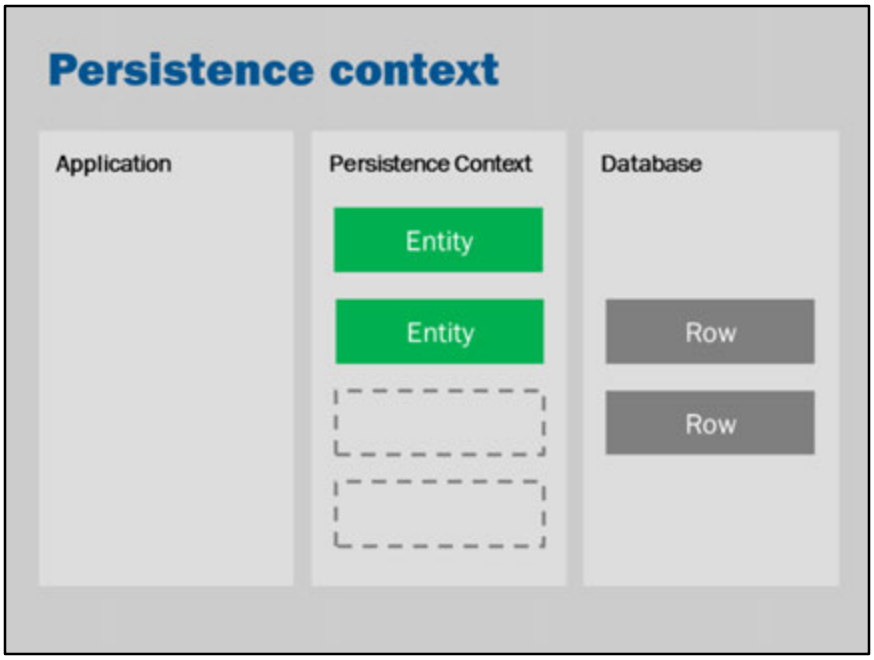


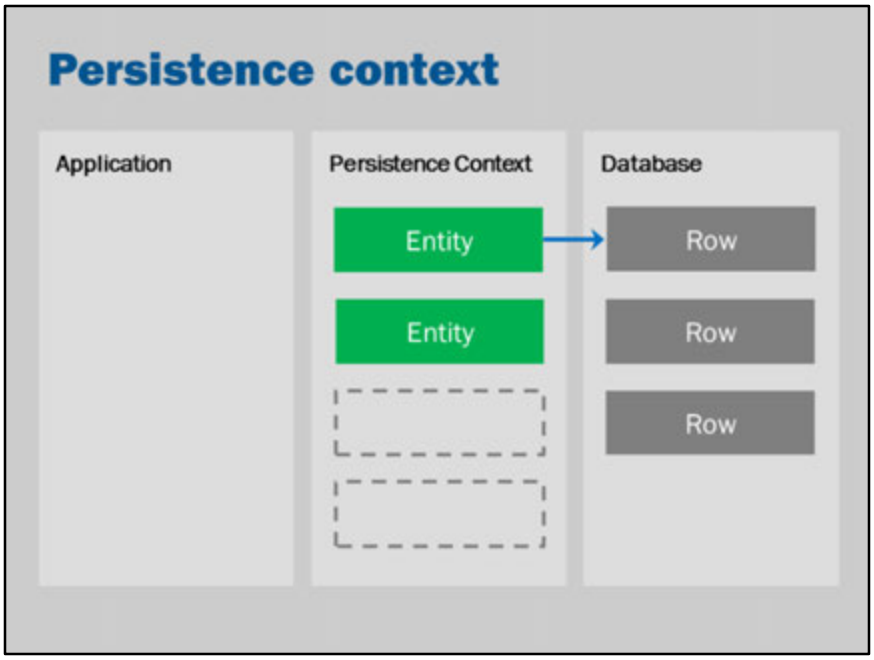
Calling find will retrieve that object from the in-memory list.


Persistence context
Application
Persistence Context
Database
Entity
Row
Row


Persistence context
Application
Persistence Context
Database
Entity
Entity
Row
Row

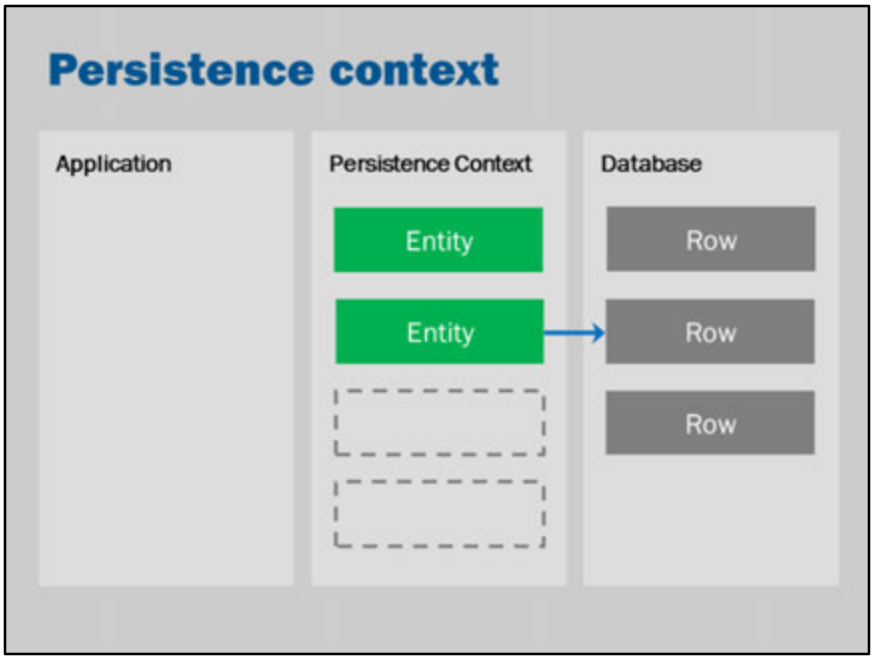

Persistence context
Application
Persistence Context
Database
Entity
Entity
Row
Row


Persistence context
Application
Persistence Context
Entity
Entity
Database
Row
Row

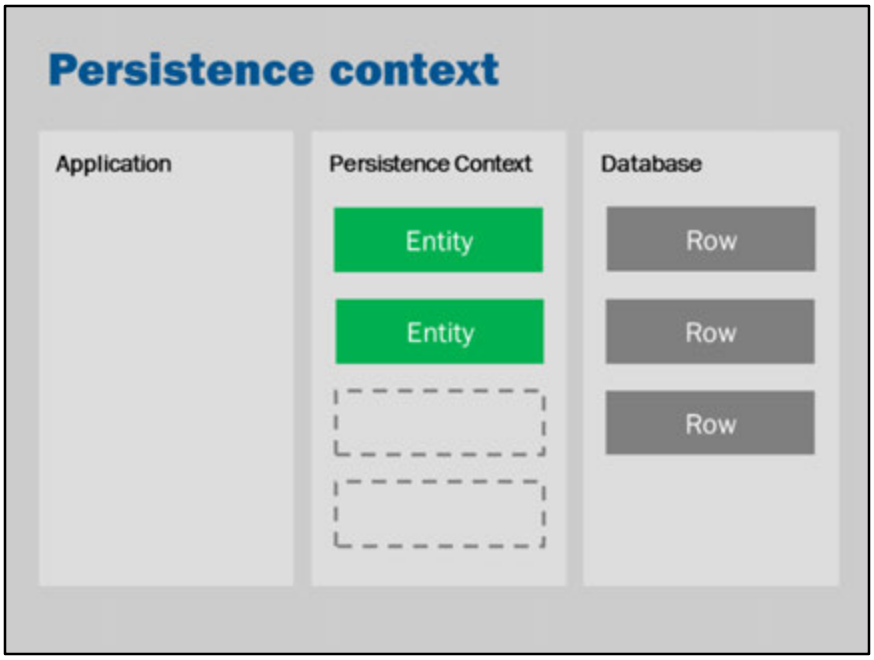


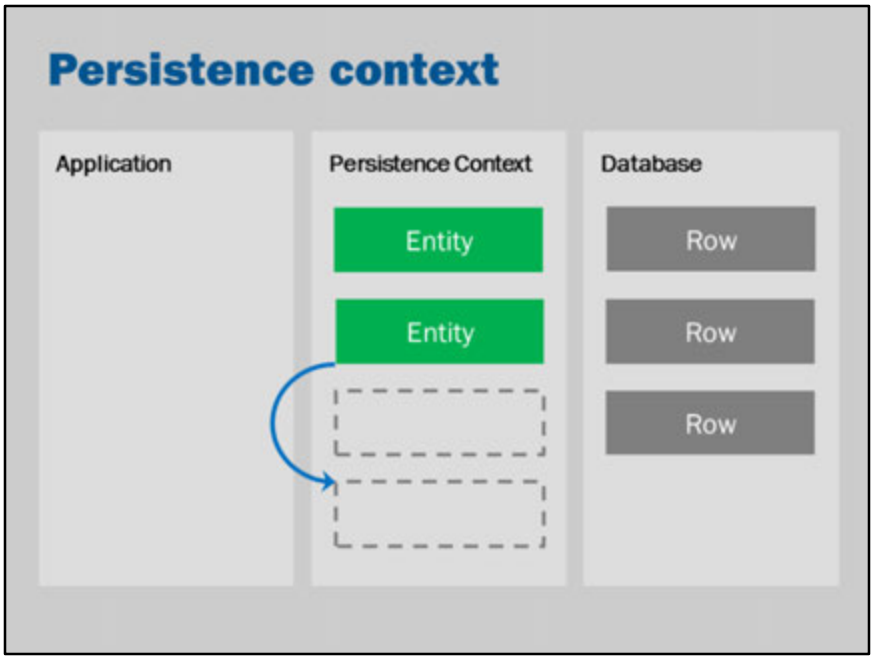
The changes to the in-memory database aren't saved until you call flush or the transaction completes (i.e., your EJB method finishes execution).

When you call flush, the new objects in the in-memory list are then inserted into the database (SQL: INSERT INTO).

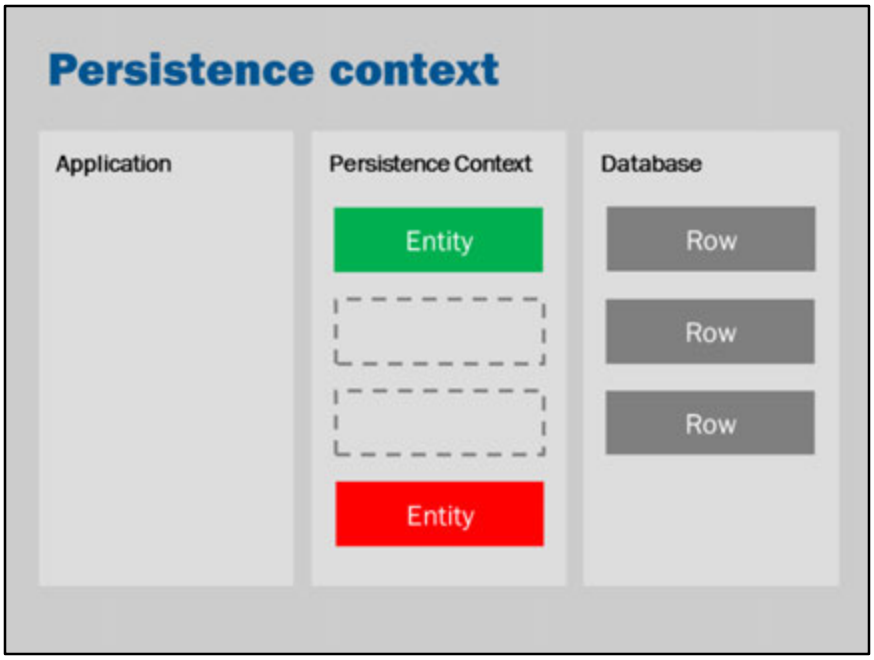


When you call flush, if you've modified any objects in the in-memory list, then those changes are also saved back into the database (SQL: UPDATE).

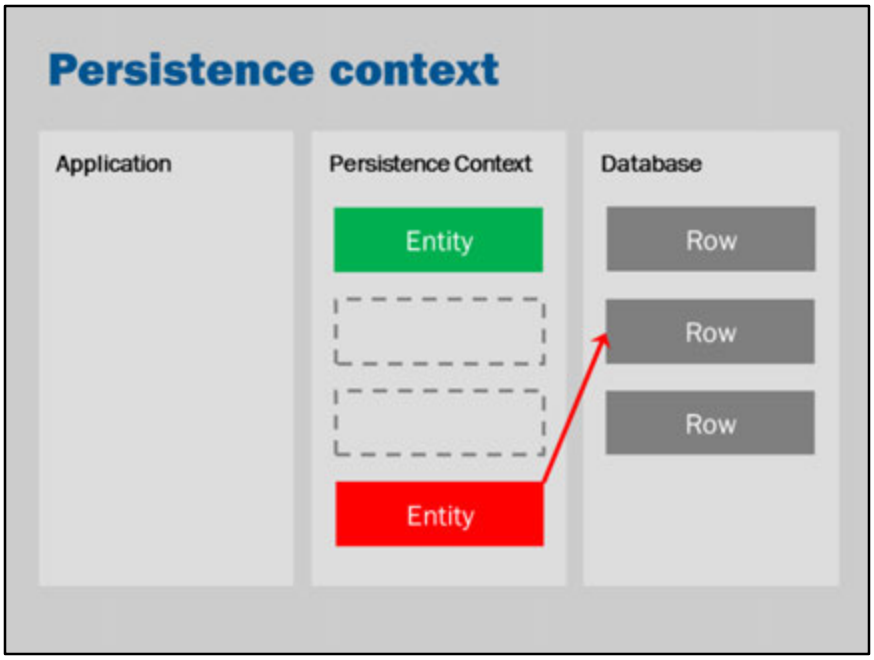

Persistence context
Application
Persistence Context
Entity
Entity
Database
Row
Row
Row



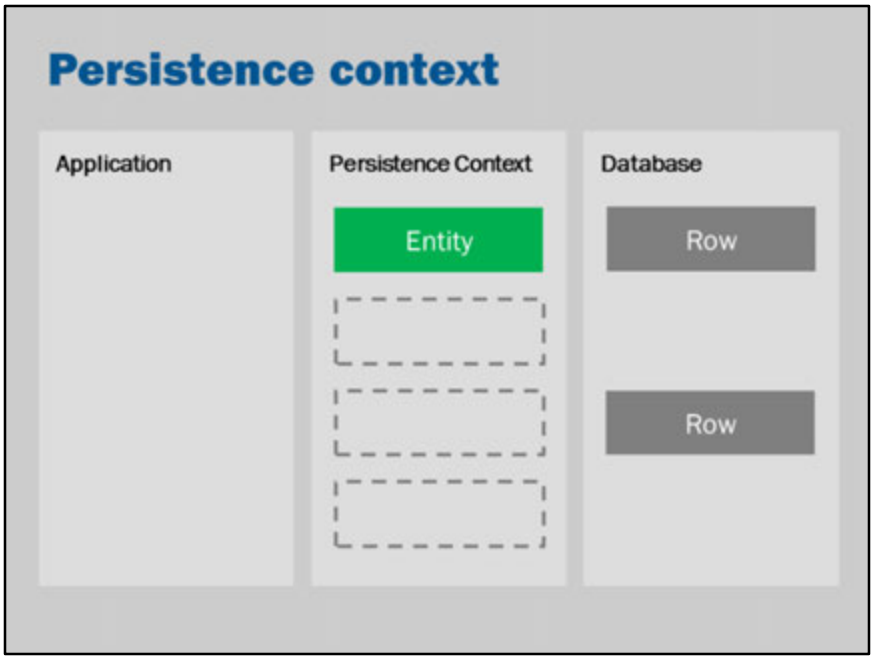
When you insert an object, it isn't saved immediately (it waits until the flush or transaction completion).
In the same way, when you delete an entity, it doesn't get removed from the database immediately.

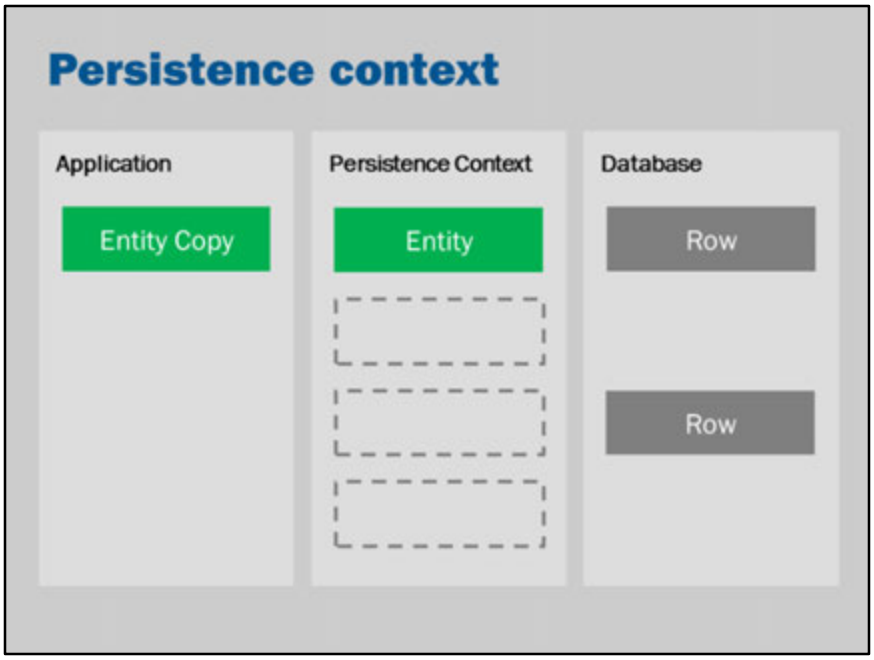


Instead, the deleted entity stays is remembered in an in-memory list of entities to delete.



When the changes are flushed to the database, then JPA will execute the SQL statement that performs the deletion (SQL: DELETE FROM)


Persistence context
Application
Persistence Context
Entity
Database
Row
Row



Suppose you have an object in the in-memory database, and an object in memory with an identical primary key.

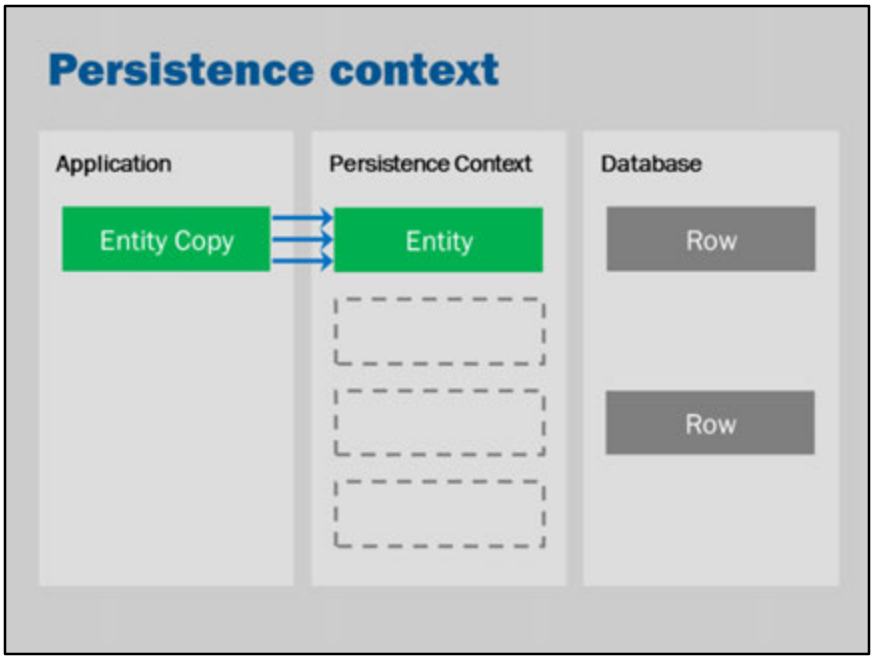
If you want to make the values the same, one way to do this would be to find the entity in memory, then copy across all the values:

e.g.,

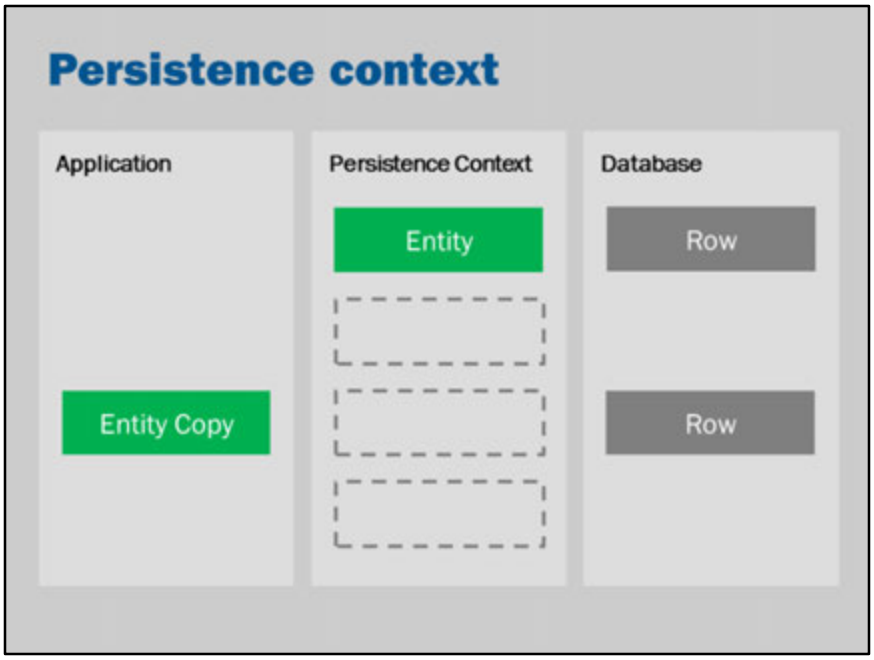
MyEntity copy = ….;

MyEntity current = em.find(copy.getId());
current.setName(copy.getName());
current.setAge(copy.getAge());
current.setAddress(copy.getAddress());
current.setSalary(copy.getSalary());

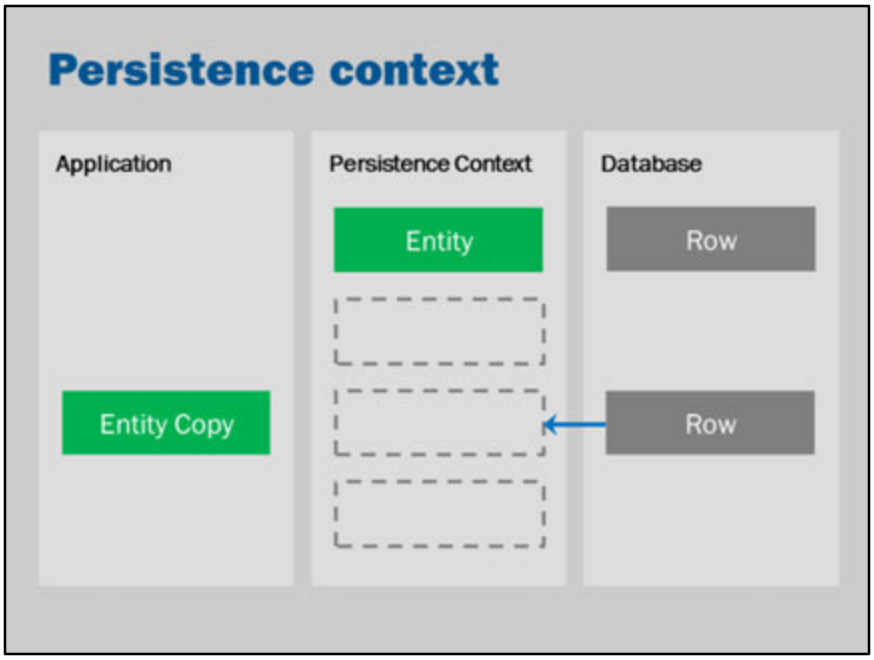
However, JPA provides the merge function that does the same thing.

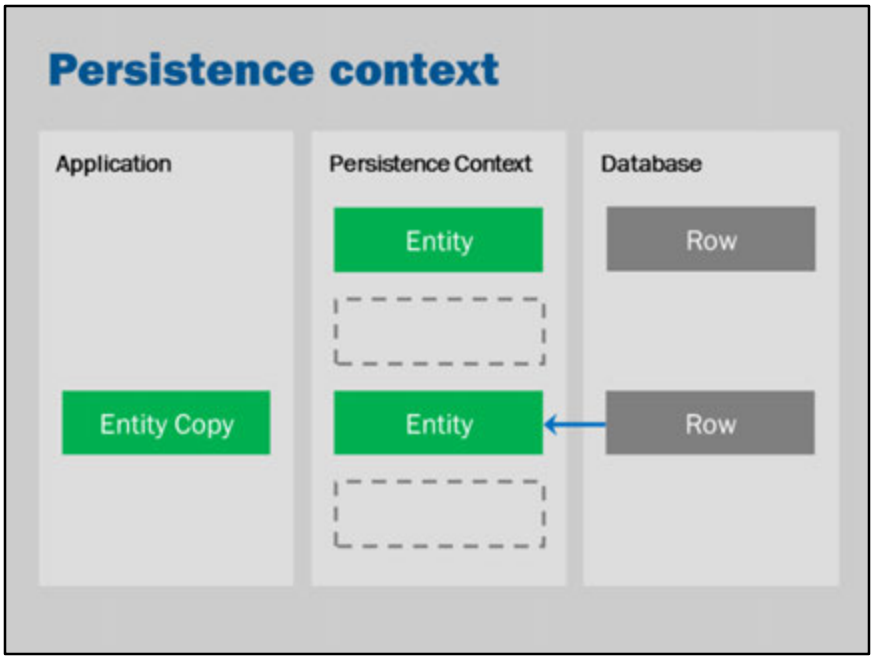


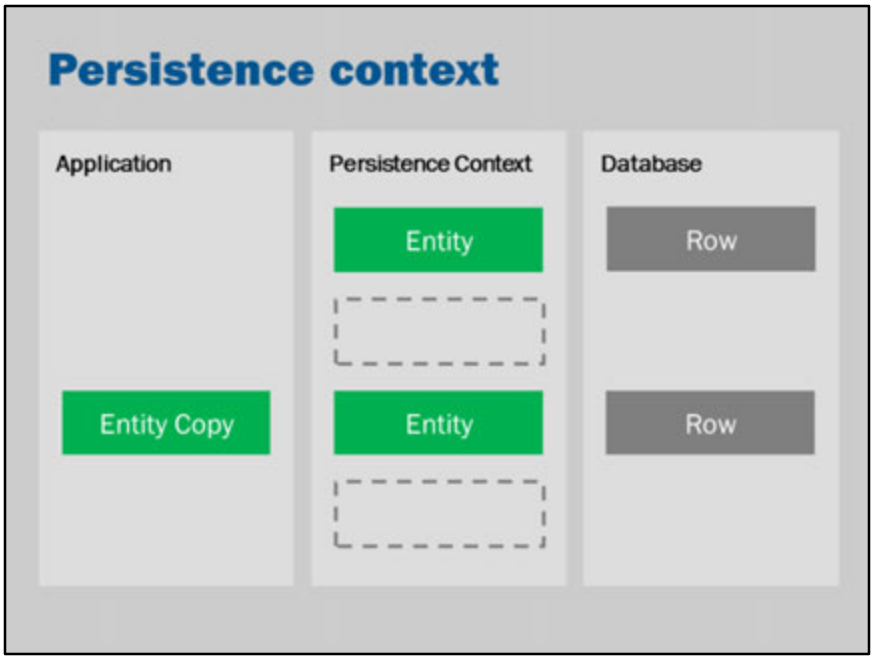
Merge will find the object in the in-memory list, and then copy across all the properties.

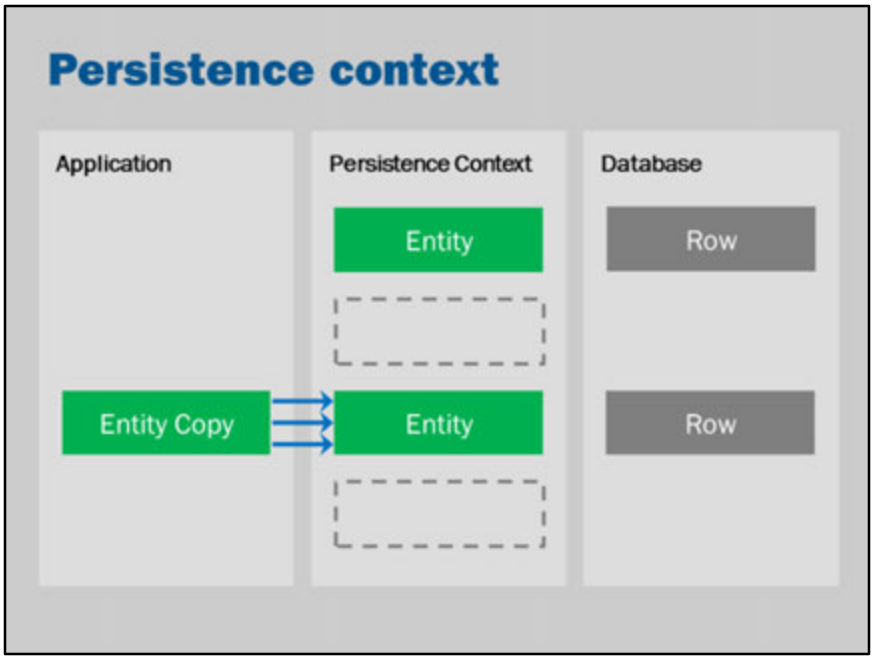


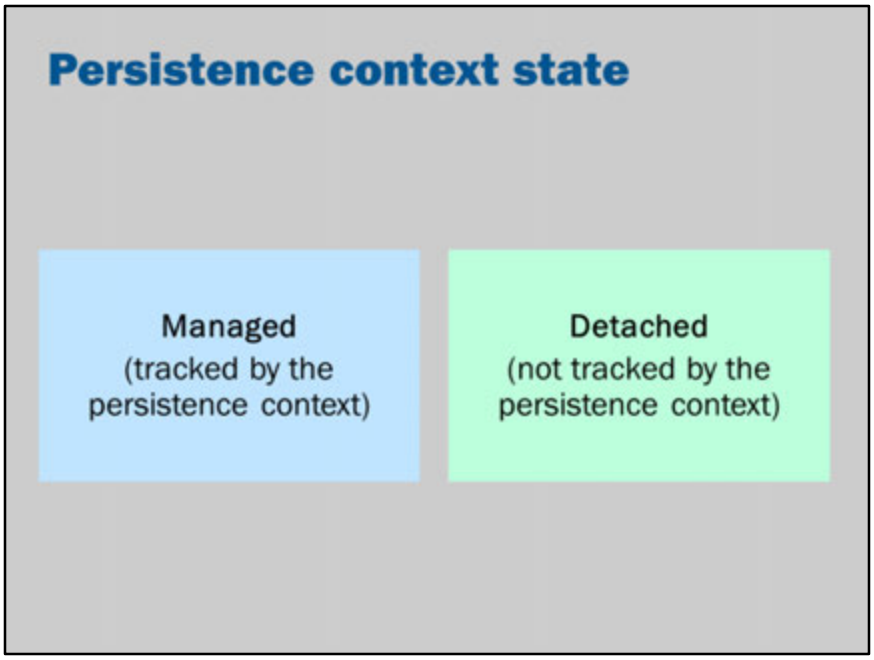
If there is no matching object in memory, then JPA will first retrieve the entity from the database and then copy across the values.
(So it still works like a find and then a copy).


Persistence context
Application
Persistence Context
Database
Entity
Row
Entity Copy
Row


Persistence context
Application
Persistence Context
Database
Entity
Row
Entity Copy
Entity
Row


Persistence context
Application
Persistence Context
Database
Entity
Row
Entity Copy
Entity
Row


Persistence context
Application
Persistence Context
Database
Entity
Row
Entity Copy
Entity
Row



A persistence context is a collection of managed entities.
In a given persistence context, there is no more than one unique entity instance for each identity (i.e., a database row will have at most one instance in a persistence context).
The persistence context is managed by the EntityManager.

The best way to think of a persistence context is as a kind of cache.
It keeps track of the connections between the database and the entities.

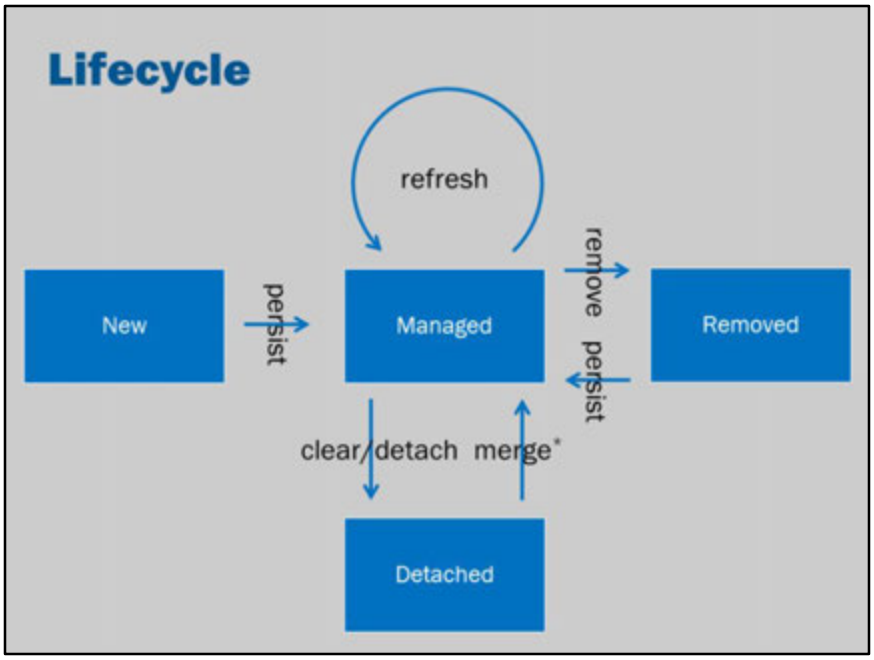
Objects that are in the persistence context will be tracked.
Objects that are in the persistence context are said to be managed.

If you make a change to a managed objects, they will be saved to the database when the persistence context is flushed.
The context can be flushed on demand (by calling em.flush()) and it will also be flushed automatically at the end of a method/transaction if you use JPA from within an EJB.

If you detach an object from the persistence context, there might still be a

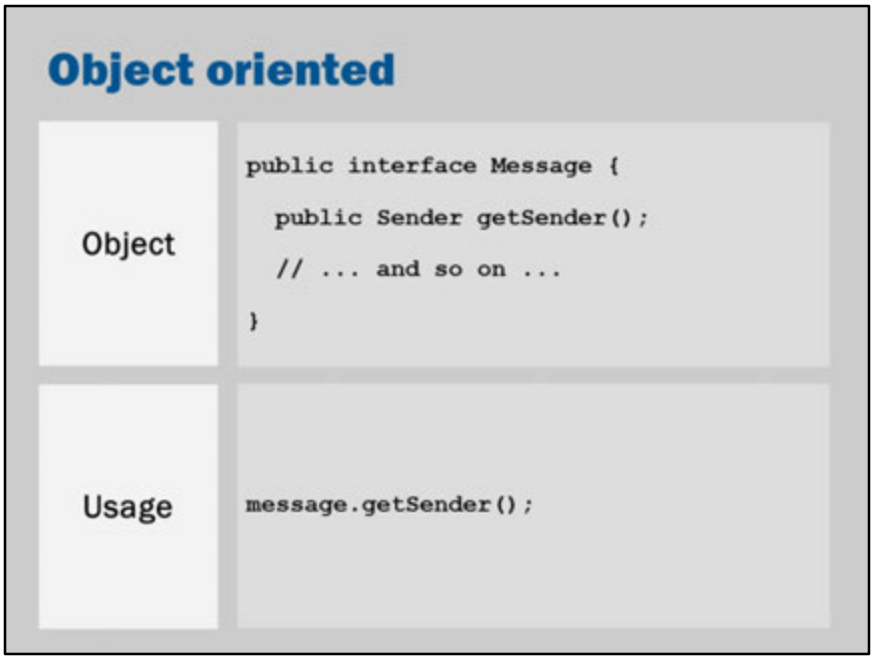
relationship between the database and the object. However, it is no longer connected to the persistence context. This means that any change you make to the detached entity will not be reflected in the database.

If you create a new instance of an Entity, then it is not yet associated with the database. It isn't until you call em.persist that the entity is associated with the persistence context. It will be saved to the underlying database when the persistence context is flushed.



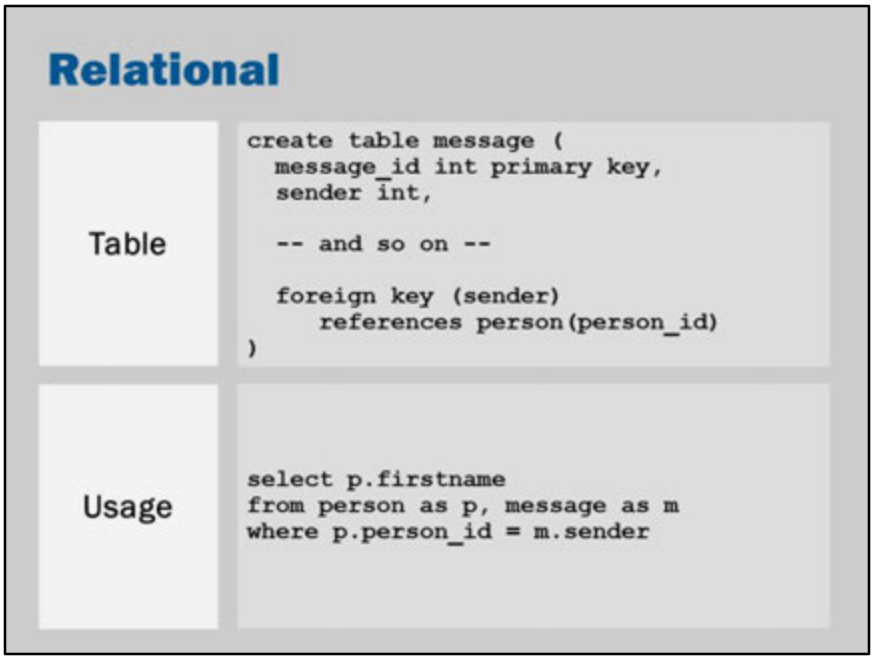
This illustrates the lifecycle of an entity in a persistence context.
Only changes to managed objects will be saved to the database.
A removed object is a still associated with the persistence context (so it isn't detached), but it will be removed from the database when the persistence context is flushed.

Relationships



In an object-oriented design, the properties of objects are other objects.

So, if you have a Message with a sender, then in an object-oriented design, calling getSender() should return a sender Object.



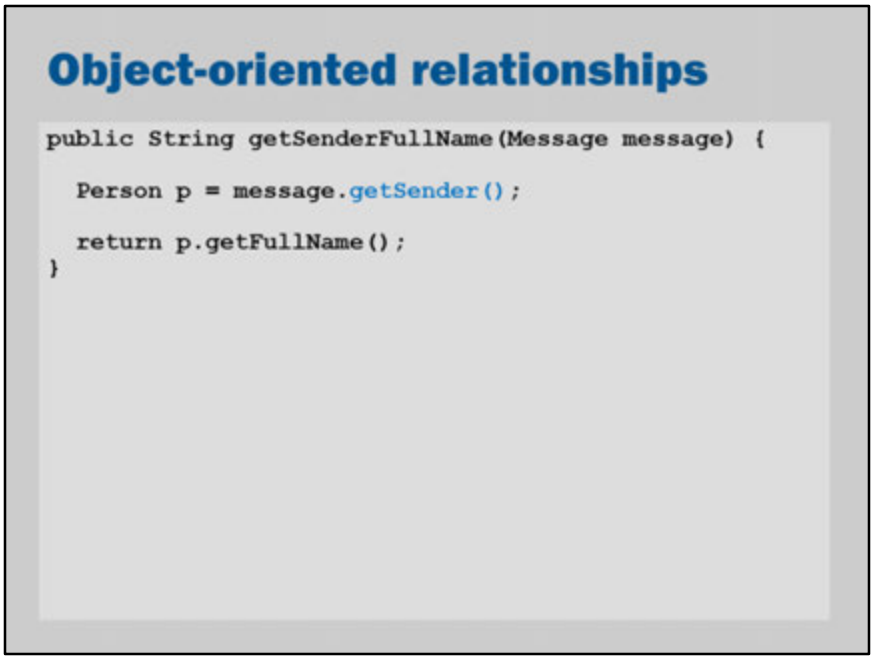
Relational databases don't quite have the same concept.
Instead of directly accessing properties, in a relational database you join two separate tables.

Manual relationship navigation

```
public String getSenderFullName(Message message) {

  int id = message.getSenderId();
  Person p = entityManager.find(Person.class, id);

  return p.getFullName();
}
```

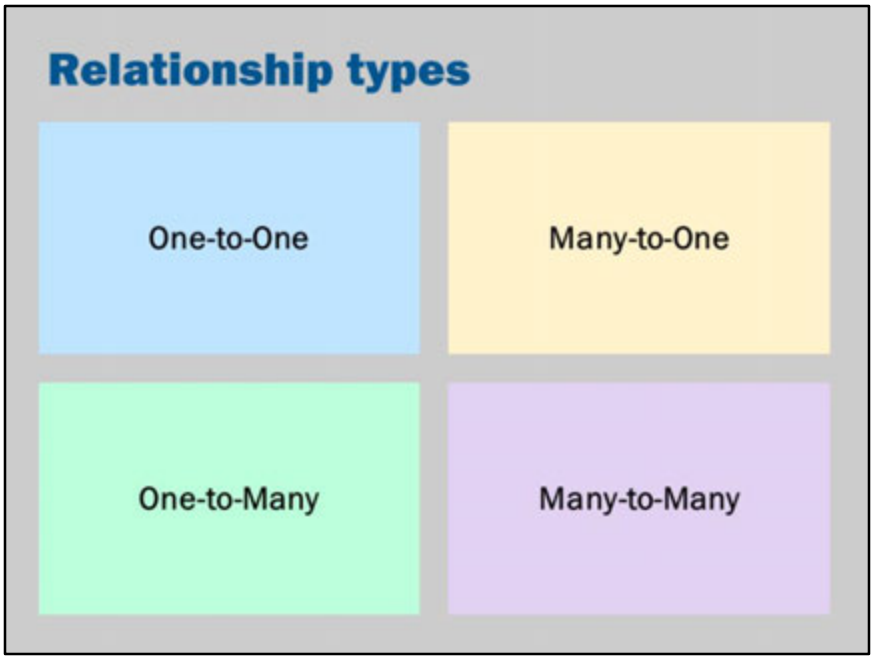
You can use JPA in a relational-database style.
i.e., get the foreign key, and then look up the other object using the foreign key as the primary key.

This works, but it is clumsy and not very object-oriented.



What we would ideally like is the ability for JPA do automatically handle the join so that it acts like ordinary object-oriented code even though, behind the scenes, there's a join that has happened.

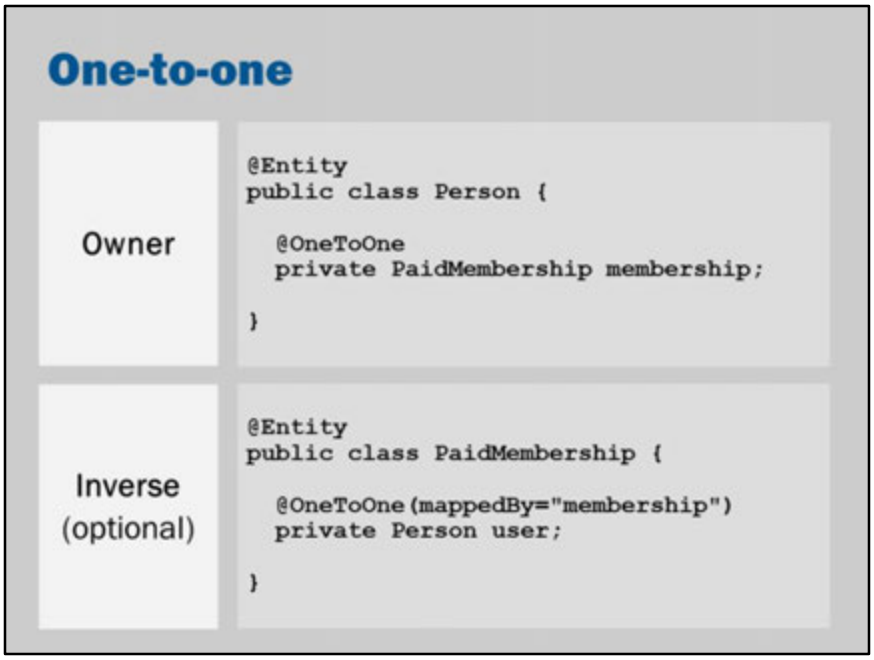
Fortunately, this is what JPA can do.



What are some examples of these types of relationships?

In our social network:

- One-to-one: A user might have at most one paying account and a paying account would have at most one user (actually a user might have zero or one paying accounts: zero if they're a "free" user and one if they're a paying user)
- Many-to-one: a message might have one sender, but those senders might have sent many messages
- One-to-many: a user might have sent many messages, but a message will only have one sender
- Many-to-many: a message might have many recipients, and a user can be the recipient of many messages



JPA supports the definition of relationships.

In JPA there is a concept of an "owner" and the "inverse" side of a relationship.

This distinction is important because:

- The "owning" side is what JPA uses to do the database updates. If you change the inverse side, but not the owning side, then JPA will not (necessarily) update the underlying database.
- If you change one side, you are responsible for also updating the other side to ensure it is consistent.

The inverse side is identified by the "mappedBy" attribute of the relationship annotation.
Note: the mappedBy attribute refers the **field/property** of the other class (NOT the underlying database column).

Many-to-one

Owner

```
@Entity
public class Message {

  @ManyToOne
  private Person sender;

}
```

Inverse
(optional)

```
@Entity
public class Person {

  @OneToMany(mappedBy="sender")
  private List<Message> sentMail;

}
```

A OneToMany relationship is just the opposite side of a ManyToOne relationship.

If you want a bidirectional relationship (i.e., the owner relationship AND the inverse relationship), then the ManyToOne side of the relationship must always be the owner.

If you don't want a bidirectional relationship (i.e., you don't want the inverse side), then it is possible to use the OneToMany annotation on its own. In this case, the OneToMany annotation is the owner because there is no inverse relationship.

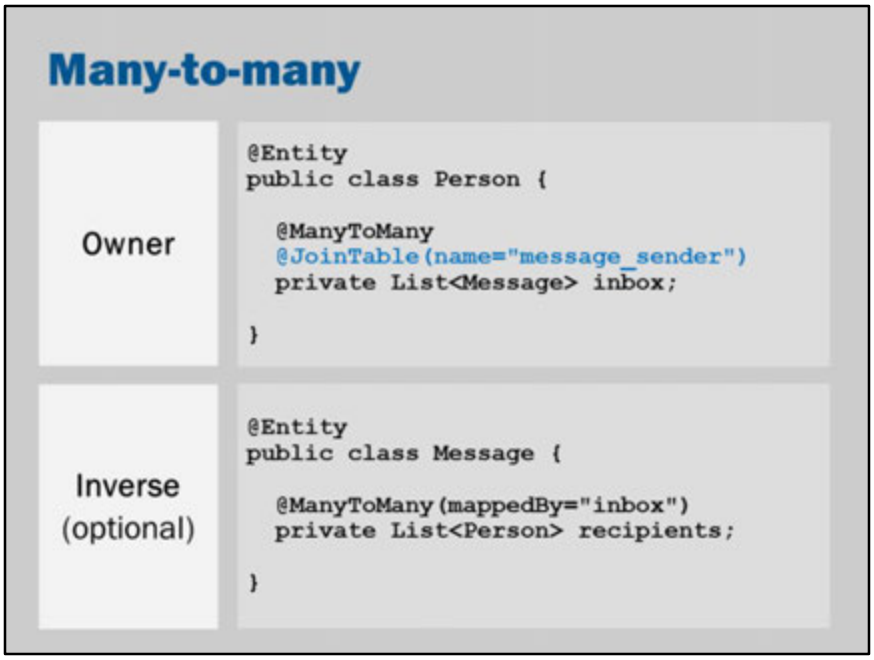
Many-to-many

Owner

```
@Entity
public class Person {

  @ManyToMany
  private List<Message> inbox;

}
```

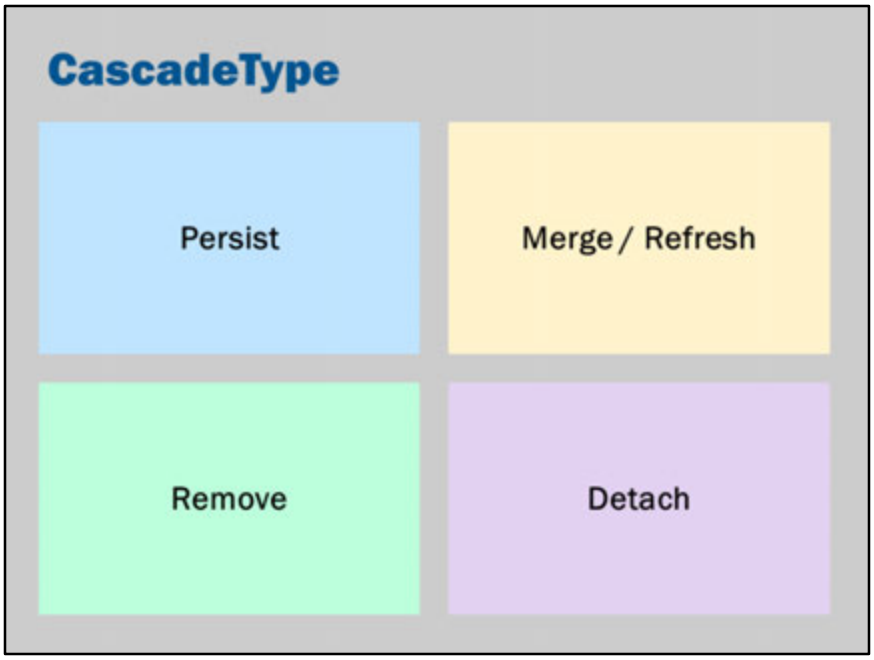
Inverse
(optional)

```
@Entity
public class Message {

  @ManyToMany(mappedBy="inbox")
  private List<Person> recipients;

}
```



One-to-one, many-to-one (and conversely, one-to-many) relationships can be defined using an extra column on underlying tables.
However, many-to-many relationships must be defined using separate tables.
These tables are called join tables.
JPA will automatically create such tables for you.
However, if you want to override the default name of the table, you can do so with with @JoinTable annotation.

When you define a relationship, you can use any of these collection types.
The Set collection type is probably closest to the semantics of relational databases.
However, in practice, List is more widespread throughout Java and so it is more commonly the collection used with JPA as well.



Relationships may be defined with a different cascade type.

A one-to-many relationship may be defined to cascade all updates using a declaration as follows:
@OneToMany(cascade=CascadeType.ALL)

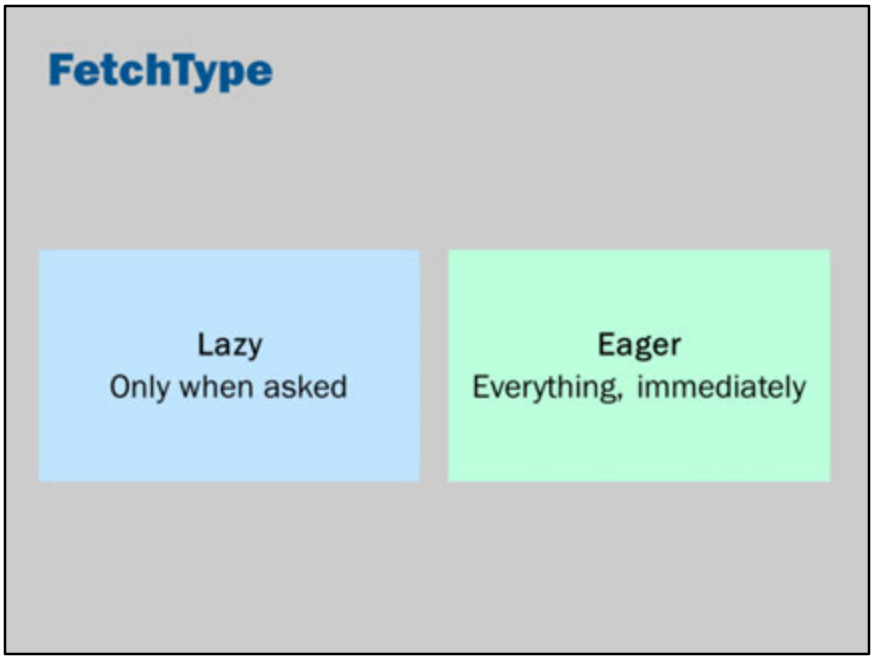
You might create an Entity for a person and also add to its relationships a number of additional entities for their office and home address.
Cascade tells JPA that when you persist the person entity, it should also "cascade" that persist operation.
All of the new office and home address entities in the relationship should also be persisted.
The change applies to all the related objects.

If you don't have cascade, then you need to manually persist each of the related objects to the entity manager as well.

The default is to not cascade.
In practice, it is probably better to not cascade changes as this can result in

unexpected bugs.

You might use cascade when you have entities that are fully "dependent" on something else.
For example, if you track the number times you call a customer, you might make the relationship between a customer and the phone calls a cascade relationship. This is because you're unlikely to need to treat the phone call as an important entity in its own right.
You would only be referring to the phone calls in reference to the customer.



By default, JPA is "lazy".
If it is in a relationship with other entities, it won't query the database to fetch the related entities until it has to.
That is, it will not query the database until you actually start using the relationship field/property of your entity.

You can override this default behavior.
You can tell JPA to load the data immediately.

@OneToMany(fetch = FetchType.EAGER)

What are some advantages and disadvantages of the two approaches?
"Lazy" avoids the need for additional database queries, especially if they aren't needed.
"Eager" makes the data available faster and has the potential for being more efficient than lazy approaches if you know, in advance, that the relationship data will be needed.

When you're writing JPQL queries you can also specify that particular relationships

should be retrieved eagerly during the query.

Advanced note:
You can also use a fetch-join in JPQL to eagerly retrieve a relationship in a particular query:

From the JPA specifications, section 4.4.5.3:

A FETCH JOIN enables the fetching of an association or element collection as a side effect of the execution
of a query.
The syntax for a fetch join is
fetch_join ::= [**LEFT** *[***OUTER***]* | **INNER** *]* **JOIN FETCH** *join_association_path_expression*
The association referenced by the right side of the FETCH JOIN clause must be an association or element
collection that is referenced from an entity or embeddable that is returned as a result of the query.
It is not permitted to specify an identification variable for the objects referenced by the right side of the
FETCH JOIN clause, and hence references to the implicitly fetched entities or elements cannot appear
elsewhere in the query.
The following query returns a set of departments. As a side effect, the associated employees for those
departments are also retrieved, even though they are not part of the explicit query result. The initialization
of the persistent state or relationship fields or properties of the objects that are retrieved as a result
of a fetch join is determined by the metadata for that class—in this example, the Employee entity
class.
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
A fetch join has the same join semantics as the corresponding inner or outer join, except that the related
objects specified on the right-hand side of the join operation are not returned in the query result or otherwise
referenced in the query. Hence, for example, if department 1 has five employees, the above query

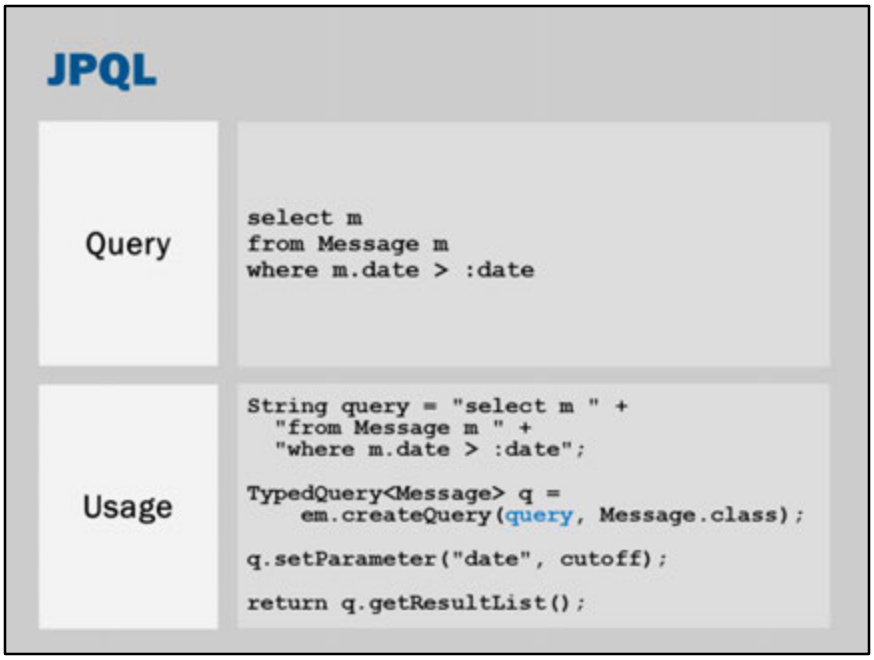
returns five references to the department 1 entity.
The FETCH JOIN construct must not be used in the FROM clause of a subquery.

Key points

JPA supports one-to-one, one-to-many, many-to-one and many-to-many relationships:

- One entity needs to be the owner of the relationship
- If you declare the relationship on the other entity, then it needs a mappedBy element on the relationship annotation

JPQL



If we are using JPA, we may not know the precise mapping that is used to translate entities into database tables.
Thus, if we want to query the database, we cannot use SQL because we do not know the underlying database structure.

To solve this problem, JPA provides JPQL.
JPQL is a query language, that looks very similar to SQL.
JPQL queries allow you to refer to the attributes of entities in order to perform a database query.

For example, if we have an entity named "Message" with a field or property named "date", it might be mapped to a database table named "social_message" and a column named "date_sent".
When writing JPQL, you refer to the field/property (in the JPA specification this is referred to as an attribute) name rather than the underlying database name.

To declare parameters, you do not use a question mark ("?") like we did with PreparedStatements in JDBC.
Instead, you use a colon (":") before a name.

Named queries

Query

```
@Entity
@NamedQuery(
  name="findMessageAfterDate",
  query="select m from Message m " +
        "where u.date > :date")
public class Message
  implements Serializable {

  ...
}
```

Usage

```
TypedQuery<Message> query =
  em.createNamedQuery(
    "findMessageAfterDate ",
    Message.class
  );

query.setParameter("date", cutoff);

return q.getResultList();
```

Named Queries are the JPQL equivalent of prepared statements.
Because a Named Query is declared before execution, the JPA provider is able to prepare or precompile the query for performance.

A Named query must be defined on an @Entity class.

Multiple named queries

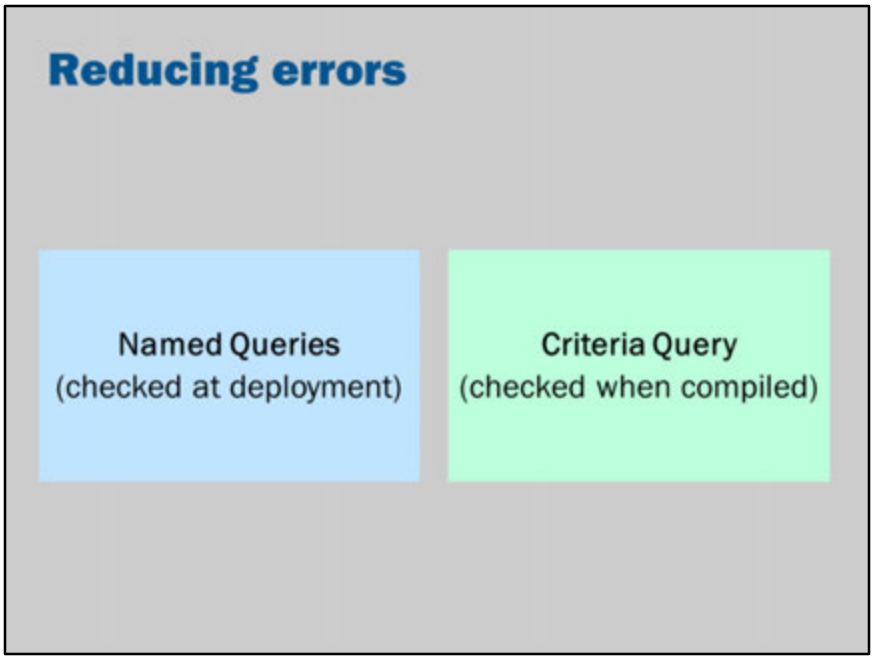
```
@Entity
@NamedQueries({
  @NamedQuery(
    name="findMessageAfterDate",
    query="select m from Message m " +
          "where m.date > :date"),
 @NamedQuery(
    name="findMessageWithStatus",
    query="select m from Message m " +
          "where m.state = :status"),
})
public class Message implements Serializable {

  ...
}
```

You can also define more than one @NamedQuery on an entity.

List vs single result

List	`return query.getResultList();`
Single Result	`return query.getSingleResult();`

Range of Results	`query.setFirstResult(30);`
	`query.setMaxResults(10);`
	`return query.getResultList();`



Named queries are automatically checked at deployment: so this is a helpful way of spotting errors before they are encountered during production.

Criteria queries are a type-safe way of programmatically building JPQL queries.

Criteria query

```
// Build a criteria query returning Messages
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Message> cq =
    builder.createQuery(Message.class);

// Set up the query: m.date > cutoff
Root<Message> m = cq.from(Message.class);
cq.select(m);
cq.where(
  builder.greaterThan(
    m.get(Message_.date),
    cutoff
  )
);

// Execute query
return em.createQuery(cq).getResultList();
```

In JPA, criteria queries are a type-safe way of constructing complex queries.
Instead of writing JPQL or SQL, you build a query using method calls.

JPA can be used to create meta-data classes (metamodel).
NetBeans will do this for you automatically.
If you have an entity named Person, then NetBeans will automatically create a meta-data class named Person_ (i.e., an underscore is added to the end of the class name).

This meta-data class contains the names of columns.
You can use this meta-data to create complex queries that refer to the attributes of an entity in a type-safe way.

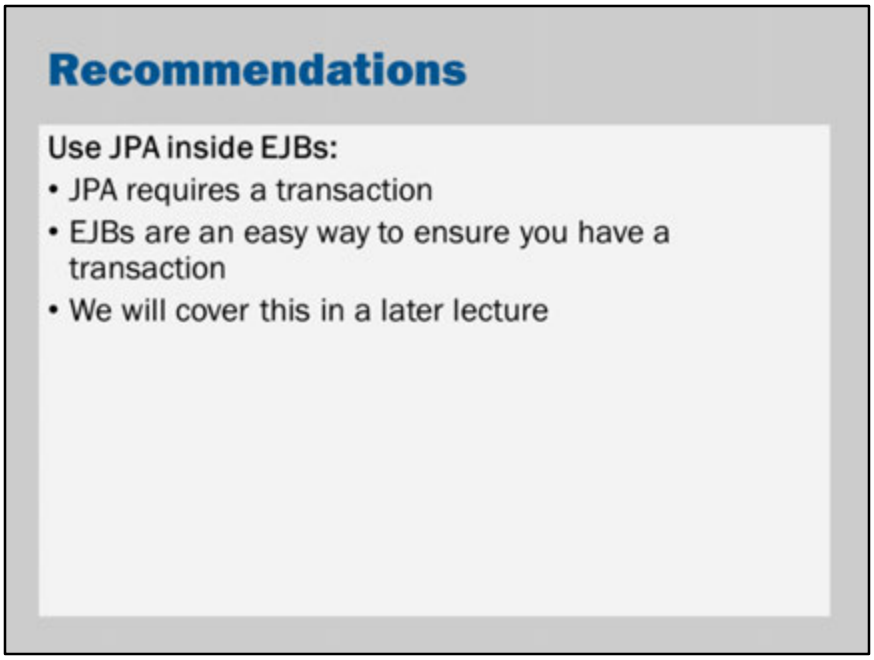
To read more about the criteria query, I recommend referring to the Beginning Java EE 7 book:
http://find.lib.uts.edu.au/?R=OPAC_b2874770

Alternately, there are some online references:
http://docs.oracle.com/javaee/7/tutorial/doc/persistence-criteria.htm
http://www.objectdb.com/java/jpa/query/criteria

Key points

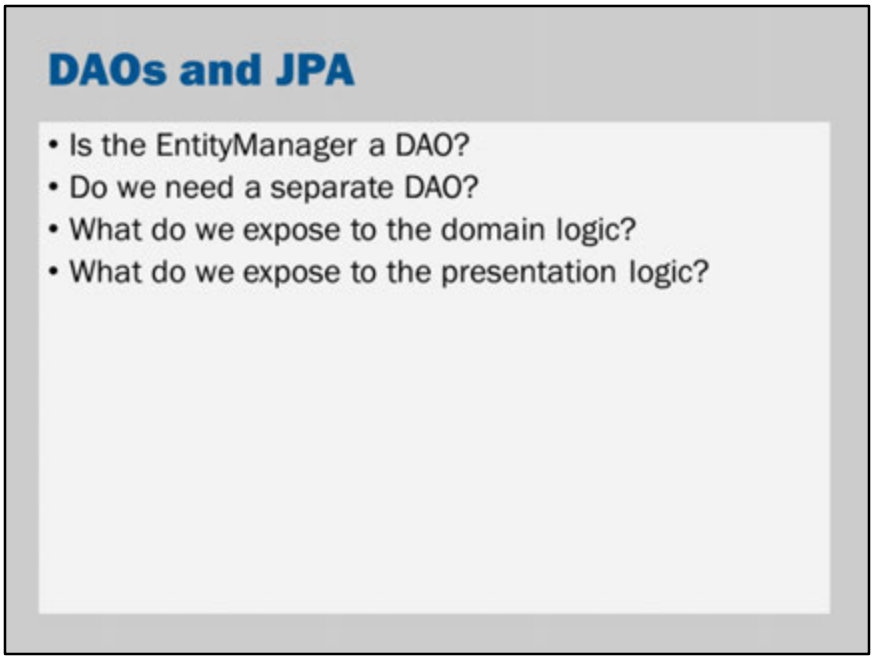
- Named queries and criteria queries are checked *prior* to execution
- Named queries are efficient and easier to understand
- Criteria queries allow for type-safe creation of dynamic queries

Bonus slides



JPA must run be used with a transaction.
For now, the easiest way to do this is by using JPA from within an EJB.
This is because Java EE automatically manages transactions inside an EJB.
We will explore transactions in a future lecture.

Ignoring transactions, there are still good reasons to only use JPA inside an EJB.
JPA is a technology that is used by domain logic.
Since domain logic is typically mapped to EJBs, this means that your JPA code will naturally be used by EJB (or called from a method in an EJB).



I have referred to the EntityManager as being very similar to a Data Access Object (DAO).

So, what do we do with DAOs?

First, why do we use DAOs?

DAOs provide an abstraction from the database.
DAOs let you substitute different databases (Java DB vs Oracle vs Microsoft SQL Server).
In fact, DAOs let you substitute different storage technologies (database vs document store vs XML documents vs filesystem).

JPA also provides an abstraction from the database.
JPA lets you substitute databases (it should automatically deal with the differences between database vendors).
There are some efforts to bring JPA to non-SQL databases (e.g., Hibernate OGM, and EclipseLink support for MongoDB and Oracle NoSQL).

So, JPA provides the abstraction and reusability of a DAO. However, DAOs are more general than JPA.

If we want to continue using DAOs, we can certainly do so.
It is very easy to implement a DAO using JPA.
Each method of your DAO would simply call the corresponding method in the EntityManager.
You could use the @Entity classes as your Data Transfer Objects.

However, it might be argued that writing a DAO is an unnecessary complexity.

Instead of using the DAO in your domain logic, you could use the EntityManager directly.
This "streamlines" the business logic and reduces the amount of complexity and useless "boilerplate".

However, you certainly would not use the EntityManager inside your presentation logic.
This is not only a matter of good application design and layer, but also one of ensuring that the EntityManager is running in a transaction (it does not make sense for the presentation logic to be setting up transactions, dealing with the EntityManager and choosing when to flush to the database).
If you want to expose CRUD-style operations to your presentation logic, then these should be services provided by the domain logic.
The domain logic would be implemented in an EJB and the domain logic would use the EntityManager.
If the domain logic is simple, then the CRUD operations might be exposed using a simple "Service Façade" (see the Week 10 labs).

This is a subtle question. There is no single "correct" answer. It will depend on the application and is a matter of finding a good design.

For more information refer to:
Bien, A. (2012) *Real World Java EE Patterns: Rethinking Best Practices*. p 259
or online discussion:
http://stackoverflow.com/questions/3818589/java-ee-architecture-are-daos-still-recommended-when-using-an-orm-like-jpa-2
http://www.infoq.com/news/2007/09/jpa-dao

http://www.adam-bien.com/roller/abien/entry/jpa_ejb3_killed_the_dao