Chapter -32

Introduction to the Java Persistence API

1. Java Persistence consists of four areas:

■The Java Persistence API

■ The query language

■ The Java Persistence Criteria API

■ Object/relational mapping metadata

1. Requirements for Entity Classes

■ The class must be annotated with the javax.persistence.Entity annotation.

■ The class must have a public or protected, no-argument constructor. The class may have

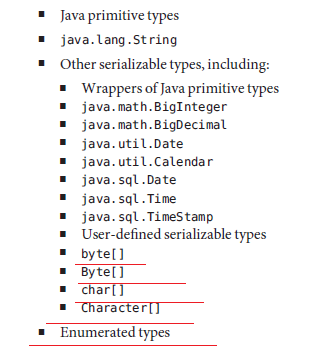
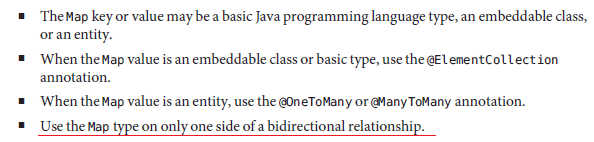
other constructors.

■the class must implement the **Serializable interface**

■ The class must not be declared final.No methods or persistent instance variables must be

declared final.

**Persistent Fields and Properties in Entity Classes**

1. The fields or properties must be of the following Java language types:
2. When using Map elements or relationships, the following rules apply
3. If the key type of a Map is a Java programming language basic type, use the annotation javax.persistence.MapKeyColumn to set the column mapping for the key. By default, the name attribute of **@MapKeyColumn** is of the form *RELATIONSHIP-FIELD/PROPERTY-NAME*\_KEY.

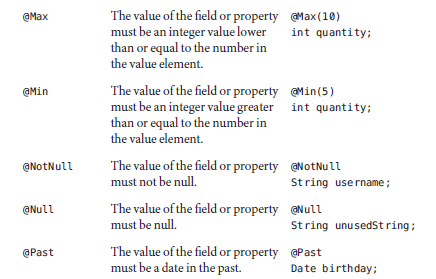
For example, if the referencing relationship field name is image, the default name attribute is

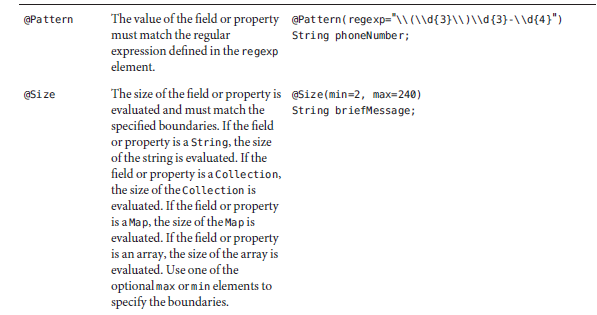
IMAGE\_KEY.

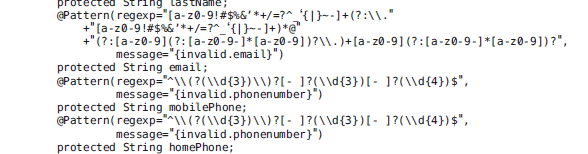
1. If the key type of a Map is an entity, use the javax.persistence.MapKeyJoinColumn annotation🡺**@MapKeyJoinColumn**
2. If the multiple columns are needed to set the mapping, use the annotation **@MapKeyJoinColumns** from javax.persistence.MapKeyJoinColumns to include multiple @MapKeyJoinColumn annotations.
3. **Needed Info on @MapKeyClass and @MapKey**

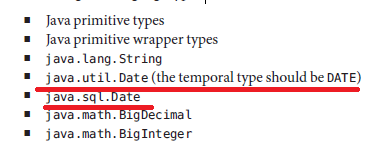
**Using Bean Validation**

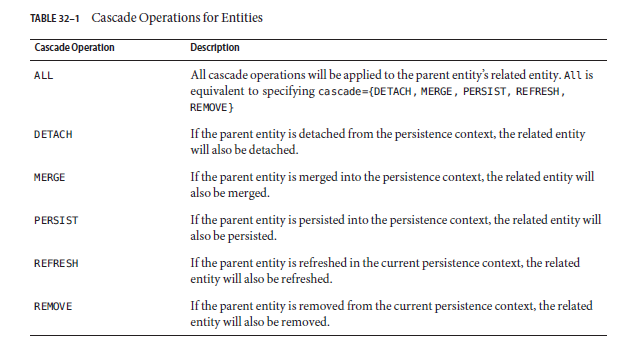
1. Validating input received from the user to maintain data integrity is an important part of application logic
2. JavaBeans Validation (Bean Validation) is a new validation model available as part of Java EE 6 platform
3. JavaBeans Validation (Bean Validation) is a new validation model available as part of Java EE 6 platform
4. Lists all the built-in constraints annotations.
5. @AssertFalse 🡺 The value of the field or property must be false
6. @AssertTrue 🡺 The value of the field or property must be true.
7. @DecimalMax🡺 The value of the field or property must be a decimal value lower than or equal to the number in the value element🡺 example 🡺
8. @Future 🡺 The value of the field or property must be a date in the future.



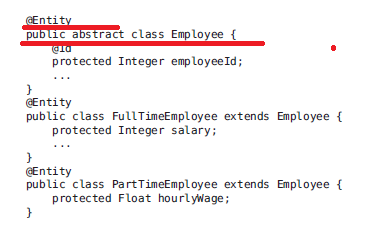


1. Some Complex Regular expression example is, 
2. In case of violation of the Java Beans Validation 🡺a validation error will be thrown.
3. The primary key, or the property or field of a composite primary key, must be one of the following Java language types

:

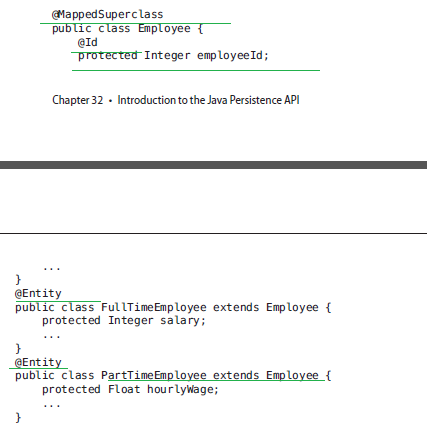
1. NEEDED TO KNOW MORE INFORMATION ON PRIMARY KEY IS AN ENTITY TYPE🡺 here notes is not prepared for this first get the basic idea and then we will come back here
2. javax.persistence.**CascadeType** 🡺 here little more info is provided 
3. Needed more info on “**mappedBy**”
4. **Orphan Removal in Relationships🡺**Page 589

**Abstract Entities**

1. An abstract class may be declared an entity by decorating the class with @Entity. Abstract entities are like concrete entities but cannot be instantiated.
2. Abstract entities can be queried just like concrete entities.

**Mapped Superclasses**

1. Entities may inherit from superclasses that contain persistent state and mapping information but are not entities. That is, the superclass is not decorated with the @Entity annotation and is not mapped as an entity by the Java Persistence provider.
2. These superclasses are most often used when you have state and mapping information common to multiple entity classes.
3. Mapped superclasses cannot be queried and can’t be used in EntityManager or Query operations. You must use entity subclasses of the mapped superclass in EntityManager or Query operations.
4. Mapped superclasses can be abstract or concrete.
5. Mapped superclasses do not have any corresponding tables in the underlying data store



**Non-Entity Superclasses🡺**I think this is same as above

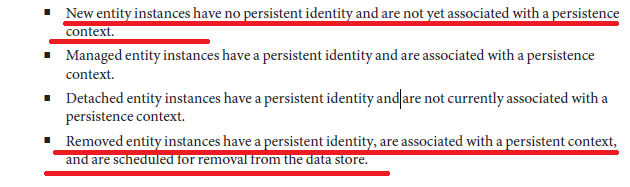
**Managing Entities**

1. Entities are managed by the entity manager, which is represented by javax.persistence.EntityManager instances.
2. **Each EntityManager instance is associated with a persistence context**
3. **A persistence context defines the scope under which particular entity instances are created, persisted, and removed**
4. **The EntityManager interface defines the methods that are used to interact with the persistence context.**
5. The EntityManager API creates and removes persistent entity instances, finds entities by the entity’s primary key, and allows queries to be run on entities.

**Container-Managed EntityManagers and Application-Managed EntityManagers 🡺Later**

**Managing an Entity Instance’s Lifecycle**

1. You manage entity instances by invoking operations on the entity by means of an EntityManager instance
2. Entity instances are in one of four states: **new, managed, detached, or removed** 🡺 but in our hibernate we saw transient, persistent and Detached states for an object



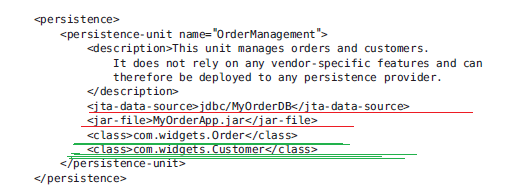
**Persisting Entity Instances 🡺** ms by persist() and CascadeType

**Synchronizing EntityData to the Database**

1. This synchronization can be achieved while commit
2. To force synchronization of the managed entity to the data store, invoke the flush method of the EntityManager instance.

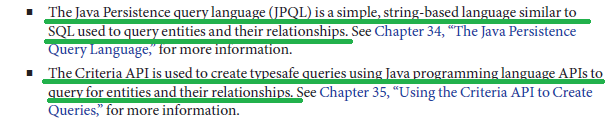
**Persistence Units**

1. **A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application**
2. This set of entity classes represents the data contained within a single data store. The jta-data-source (for JTA-aware data sources) and non-jta-data-source (for non-JTA-aware data sources) elements specify the global JNDI name of the data source to be used by the container.
3. **META-INF directory contains persistence.xml is called the root of the persistence unit.**



**Querying Entities**

1. The Java Persistence API provides the following methods for querying entities.



1. Advantage and disadvantage of JPQL
   1. **Advantage:**

* JPQL queries are typically more concise and more readable than Criteria queries.
* Developers familiar with SQL will find it easy to learn the syntax of JPQL.
* JPQL namedqueries can be defined in the entity class using a Java programming language annotation or in the application’s deployment descriptor.
  1. **Disadvantage:**
* JPQL queries are not typesafe, however, and require a cast when retrieving the query result from the entity manager. This means that type-casting errors may not be caught at compile time
* JPQL queries don’t support open-ended parameters.

1. Advantage and disadvantage of JPQL
   1. Advantage

* Criteria queries allow you to define the query in the business tier of the application
* Criteria queries are typesafe and therefore don’t require casting, as JPQL queries do
* The Criteria API is just another Java programming language API and doesn’t require developers to learn the syntax of another query language.
  1. Disadvantage
* require the developer to create several objects and perform operations on those objects before submitting the query to the entity manager.

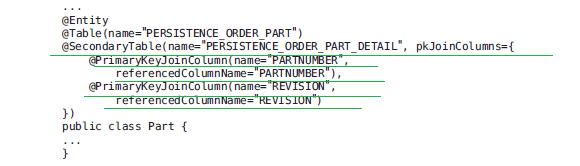
Chapter 33 🡺 Running the Persistence Examples

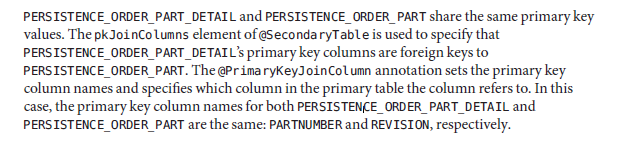
1. In this chapter a new application is developed and executed.

**Compound Primary Keys**

1. A compound primary key is made up of multiple fields and follows the requirements described in “Primary Keys in Entities”

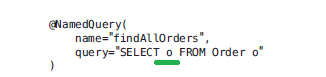
**EntityMapped to More Than One Database Table**

1. The **@Secondary** Table annotation is used to specify the secondary table.
2. 



**Using Queries 🡺**ms in our example application

1. Let’s see an example

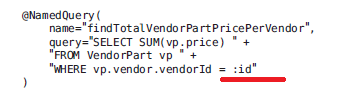


Now in the code,

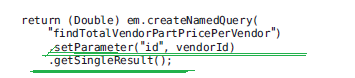


**The EntityManager.createNamedQuery method is used to run the query. Because the query returns a List of all the orders, the Query.getResultList method is used**

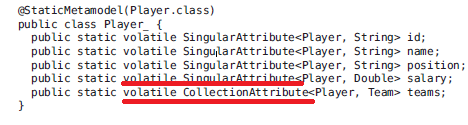
1. Example 2



When running the query, the Query.setParameter method is used to set the named parameter id to the value of vendorId, the parameter to RequestBean.getTotalPricePerVendor

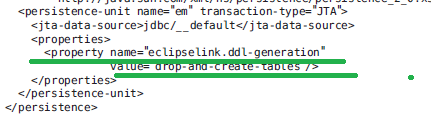


**CriteriaQueries in the roster Application**

1. **Metamodel Classes in the roster Application**
2. **Metamodel classes model an entity’s attributes and are used by Criteria queries to navigate to an entity’s attributes**
3. Each entity class in roster has a corresponding metamodel class, generated at **compile time**, with the same package name as the entity and appended with an underscore character (\_).
4. Each persistent field or property in the entity class has a corresponding attribute in the entity’s metamodel class.
5. 

**AutomaticTable Generation in the roster Application**

1. At **deployment time**, the GlassFish Server will automatically drop and create the database tables used by roster
2. This is done by setting the **eclipselink.ddl-generation** property to **drop-and-create-tables** in persistence.xml:

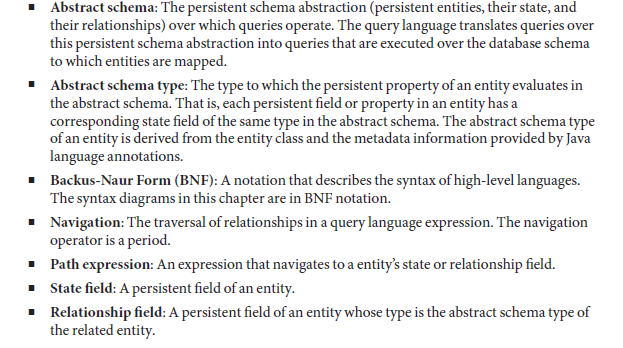


**NOTE 🡺 Automatic table creation is useful for development purposes, however, and the eclipselink.ddl-generation property may be removed from persistence.xml when preparing the application for production use.**

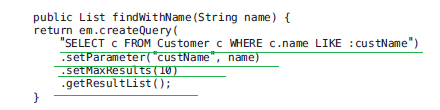
Chapter 34🡺 The Java Persistence Query Language

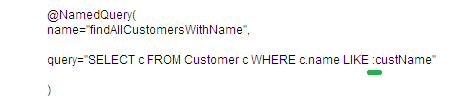
1. The Java Persistence query language defines queries for entities and their persistent state.
2. **The query language allows you to write portable queries that work regardless of the underlying data store.**

**Query Language Terminology🡺**

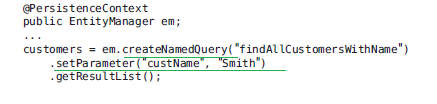


**Creating Queries Using the Java Persistence Query Language**

1. The **EntityManager.createQuery and EntityManager.createNamedQuery** methods are used to query the data store by using Java Persistence query language queries
2. The createQuery method is used to create *dynamic queries*, which are queries defined directly within an application’s business logic:
3. Ms 🡺Similarly NamedQuery is written on top of entity class not in the business logic

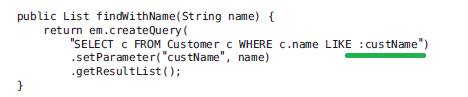


Here’s an example of createNamedQuery, which uses the @NamedQuery,



**Named Parameters in Queries**

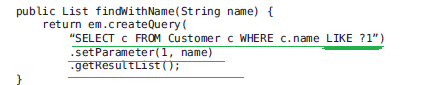
1. Named parameters are query parameters that are prefixed with a colon (:).Named parameters in a query
2. Example in case of Query🡺



**Positional Parameters in Queries**

1. Positional parameters are prefixed with a question mark (?) followed the numeric position of the parameter in the query.

for example 🡺 “SELECT c FROM Customer c WHERE c.name LIKE **?1**”

1. The **Query.setParameter(integer position, Object value)** method is used to set the parameter values.
2. 

**Simplified Query Language Syntax**

1. **Select Statements**
2. 
3. **A Basic Select Query 🡺** **SELECT p FROM Player p**
4. **EliminatingDuplicateValues 🡺** **SELECT DISTINCT p FROM Player p WHERE p.position = ?1**
5. **Using Named Parameters 🡺** SELECT DISTINCT p FROM Player p WHERE p.position = :position AND p.name = :name

**Queries That Navigate to Related Entities 🡺 Later**

**Queries with Other Conditional Expressions**

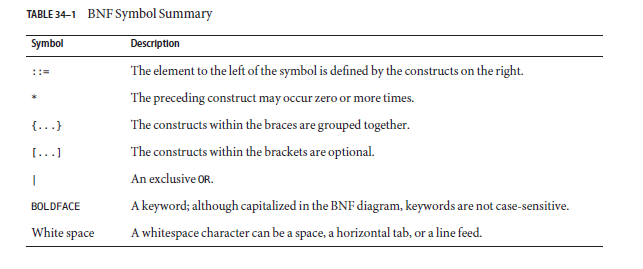
1. **The LIKE Expression 🡺** SELECT p FROM Player p WHERE p.name LIKE ’Mitch%’
2. **The IS NULL Expression** 🡺 SELECT t FROM Team t WHERE t.league IS NULL
3. **The IS EMPTY Expression**🡺 SELECT p FROM Player p WHERE p.teams IS EMPTY
4. **The BETWEEN Expression 🡺** SELECT DISTINCT p FROM Player p WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
5. **Comparison Operators**🡺 SELECT DISTINCT p1 FROM Player p1, Player p2 WHERE p1.salary > p2.salary AND p2.name = :name
6. **Bulk Updates and Deletes**
7. **Update Queries 🡺**

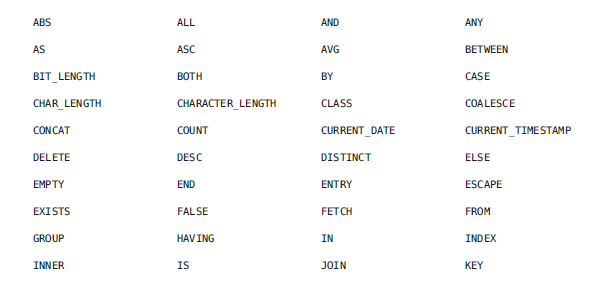
**UPDATE Player p SET p.status = ’inactive’ WHERE p.lastPlayed < :inactiveThresholdDate**

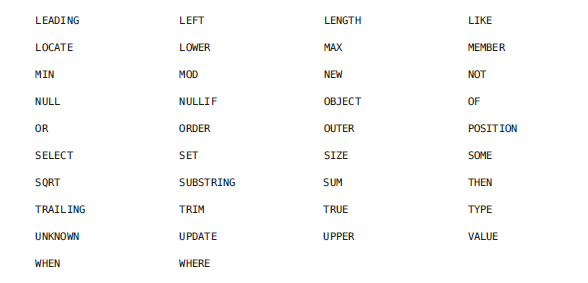
1. **DeleteQueries 🡺** DELETE FROM Player p WHERE p.status = ’inactive’ AND p.teams IS EMPTY

**Full Query Language Syntax**

1. **BNF Symbols** 🡺**we will See it Later**



1. **FROM Clause**🡺 The FROM clause defines the domain of the query by declaring identification variables.
2. **Identifiers** 🡺 An identifier is a sequence of one or more characters. The first character must be a valid first character (letter, $, \_)
3. **Keywords 🡺**ms List of keywords in JPQL is shown below

]

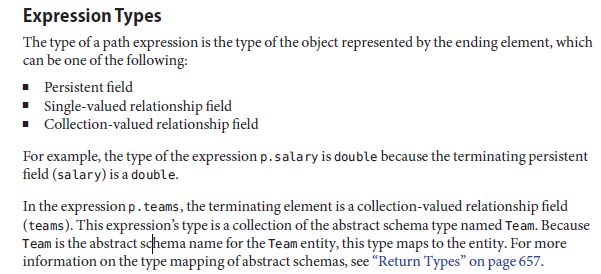
1. **Identification Variables 🡺Later**
2. **Range Variable Declarations 🡺**explanation we will come back after understanding the abstract type and etc.…
3. Example 🡺

* FROM Player p
* FROM Player AS p 🡺 A range variable declaration can include the optional AS operator
* FROM Player p1, Player p2 🡺 If the query compares multiple values of the same abstract schema type, the FROM clause must declare multiple identification variables for the abstract schema:

1. **Collection Member Declarations🡺** explanation we will come back after understanding the abstract type and etc.
2. **Joins** 🡺Explanation later

SELECT c FROM Customer c JOIN **c.orders o 🡺** WHERE c.status = 1 AND o.totalPrice > 10000

1. **Path Expressions 🡺**ms it’s like explanation for the sql statement , we will understand it once we see abstract type, schema and etc.. 🡺 there is nothing extra present in the code part for this
2. **Expression Types🡺**

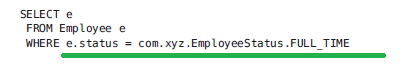
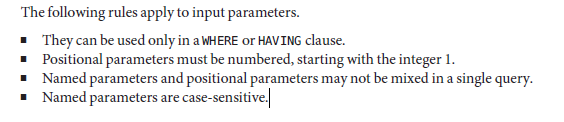


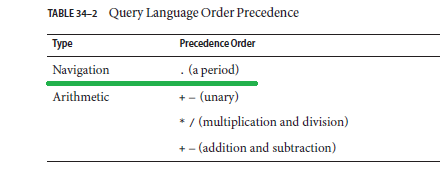
1. **WHERE Clause**

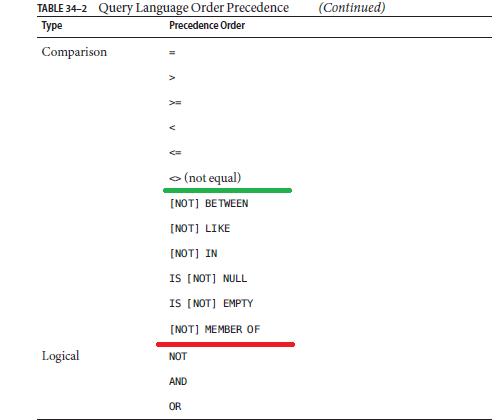
****

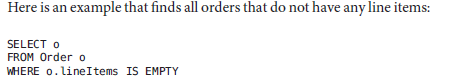
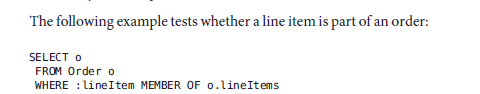
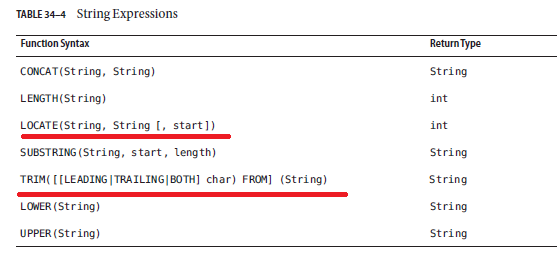
1. **Literals🡺**
2. There are four kinds of literals: string, numeric, Boolean, and enum.
3. **String literals**: A string literal is enclosed in single quotes 🡺 ’Duke’
4. **Numeric literals**: There are two types of numeric literals: exact and approximate

* An exact numeric literal is a numeric value without a decimal point, such as 65, –233, and +12.
* An approximate numeric literal is a numeric value in scientific notation, such as 57., –85.7, and +2.1.

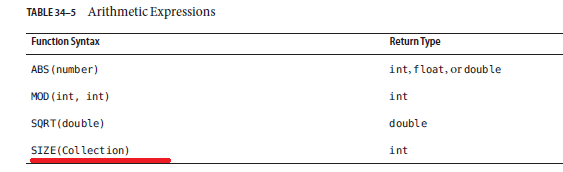
1. **Boolean literals**: A Boolean literal is either TRUE or FALSE. These keywords are not case-sensitive.
2. **Enum literals**: The Java Persistence query language supports the use of enum literals using the Java enum literal syntax, Example shown below
3. **Input Parameters🡺** An input parameter can be either a named parameter or a positional parameter.
4. **Conditional Expressions🡺**



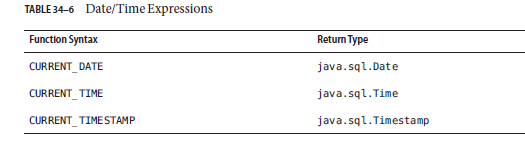
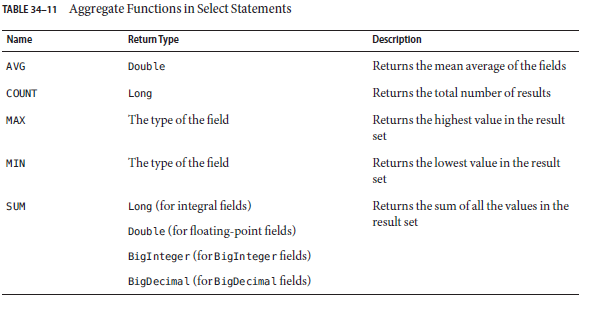
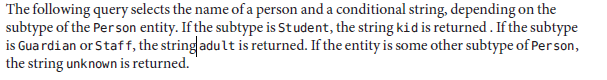


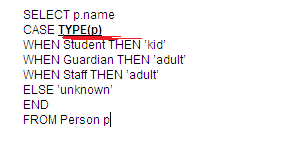
1. **Empty Collection Comparison Expressions**
2. **The IS [NOT] EMPTY comparison expression tests whether a collection-valued path expression has no elements. In other words, it tests whether a collection-valued relationship has been set.**
3. 
4. **Collection Member Expressions:**
5. **The [NOT] MEMBER [OF]** collection member expression determines whether a value is a member of a collection. The value and the collection members must have the same type.
6. 
7. **Subqueries 🡺 later**
8. **Functional Expressions**
9. The query language includes several string, arithmetic, and date/time functions that may be used in the SELECT, WHERE, or HAVING clause of a query
10. 

* The **LOCATE function** returns the position of a given string within a string.
* The **TRIM function** trims the specified character from the beginning and/or end of a string. If no character is specified, TRIM removes spaces or blanks from the string.

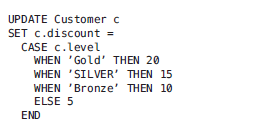
1. 

* The **SIZE function** returns an integer of the number of elements in the given collection.

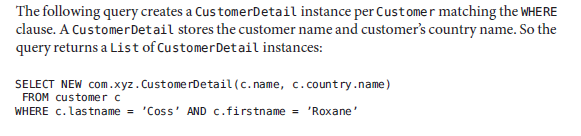
1. 
2. 
3. **Case Expressions**
   1. Example 🡺

****

* 1. Example 2 🡺



1. **Constructor Expressions**
   1. **Constructor expressions allow you to return Java instances that store a query result element instead of an Object [].**
   2. Example 🡺

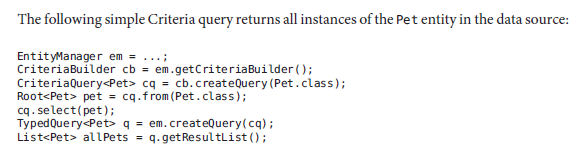


1. **GROUP BY and HAVING Clauses🡺**The HAVING clause is used with the GROUP BY clause to further restrict the returned result of a query.

Chapter 35🡺 Using the Criteria API to Create Queries

1. Criteria queries are written using Java programming language APIs, are typesafe, and are portable. Such queries work regardless of the underlying data store.

**Overview of the Criteria and Metamodel APIs**

1. Similar to JPQL, the Criteria API is based on the abstract schema of persistent entities.
2. The Criteria API operates on this abstract schema to allow developers to find, modify, and delete persistent entities by invoking Java Persistence API entity operations.
3. 
4. The equivalent JPQL query for above CriteriaQuery is🡺 **SELECT p FROM Pet p**
5. The tasks associated with each step are discussed in detail in this chapter

* To create a **CriteriaBuilder** instance, call the **getCriteriaBuilder** method on the **EntityManager instance**:



* The query object is created by using the CriteriaBuilder instance:

**Note 🡺 The query will return instances of the Pet entity, so the type of the query is specified when the CriteriaQuery object is created to create a typesafe query**.

* **The FROM clause of the query is set**, and the root of the query specified, by calling the from method of the query object:



* **The SELECT clause of the query is set by calling the select method of the query object and passing in the query root:**



* **The query object is now used to create a TypedQuery<T> object that can be executed against the data source.**



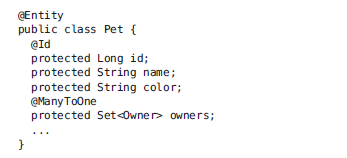
* This typed query object is executed by calling its **getResultList** method, because this query will return multiple entity instances. The results are stored in a List<Pet> collection-valued object.

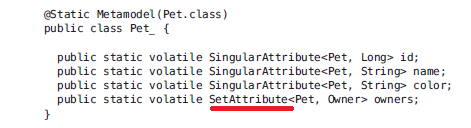


**Note 🡺 Ms the above is equivalent to the HQL 🡺 Query q = session.createQuery (“from employee”);**

**Using the Metamodel API to Model Entity Classes**

1. The metamodel class and its attributes are used in Criteria queries to refer to the managed entity classes and their persistent state and relationships.
2. Example🡺





**Using Metamodel Classes**

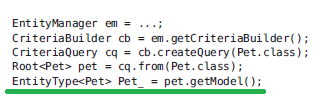
1. **Metamodel classes are typically generated by annotation processors either at development time or at runtime.**

**Note 🡺 Developers of applications that use Criteria queries may generate static metamodel classes by using the persistence provider’s annotation processor** **or may obtain the**

**metamodel class by either calling the getModel method on the query root object**

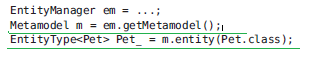
1. The following code snippet shows how to obtain the Pet entity’s metamodel class by calling

Root<T>.getModel:

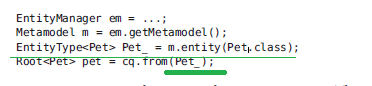


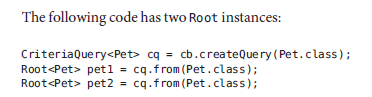
(or)

The following code snippet shows how to obtain the Pet entity’s metamodel class by first obtaining a metamodel instance by using EntityManager.getMetamodel and then calling entity on the metamodel instance.



**Using the Criteria API and Metamodel API to Create Basic Typesafe Queries**

1. **The basic semantics of a Criteria query consists of a SELECT clause, a FROM clause, and an optional WHERE clause, similar to a JPQL query. Criteria queries set these clauses by using Java programming language objects, so the query can be created in a typesafe manner.**
2. **Creating a CriteriaQuery**
3. The **javax.persistence.criteria.CriteriaBuilder** interface is used to construct,
   * Criteria queries
   * Selections
   * Expressions
   * Predicates
   * Ordering
4. To obtain an instance of the CriteriaBuilder interface, call the getCriteriaBuilder method on either an EntityManager **or an EntityManagerFactory instance.**
5. **Query Roots**
6. 
7. Criteria queries may have more than one query root. This usually occurs when the query navigates from several entities.

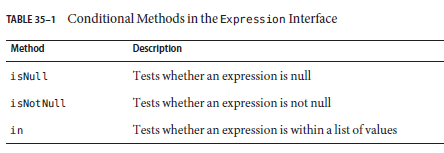


**Querying Relationships Using Joins🡺Later**

**Path Navigation in CriteriaQueries🡺** this is not much Logical thing

**Restricting CriteriaQuery Results**

1. The results of a query can be restricted on the CriteriaQuery object according to conditions set by calling the **CriteriaQuery.where** method.
2. Calling the where method is analogous to setting the WHERE clause in a JPQL query.
3. The where method evaluates instances of the **Expression interface** to restrict the results according to the conditions of the expressions.
4. **Expression instances are created by** using methods defined in the **Expression and CriteriaBuilder interfaces.**
5. The Expression Interface Methods:



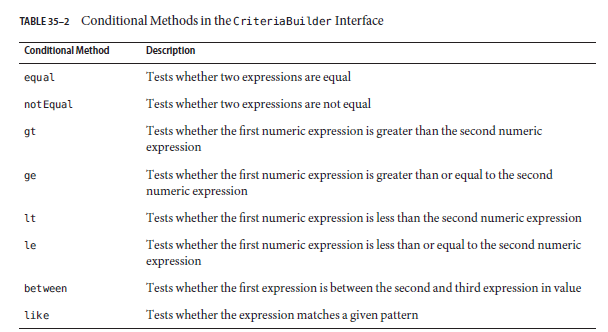
1. Example 1



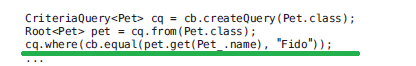
1. Example 2



1. **Expression Methods in the CriteriaBuilder Interface**



1. Now let’s see the example for each of the above methods,
   1. CriteriaBuilder.equal method:



* 1. CriteriaBuilder.gt method:

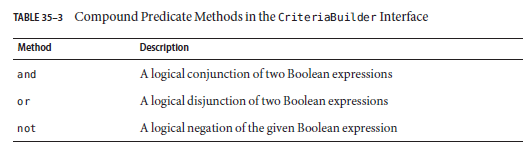
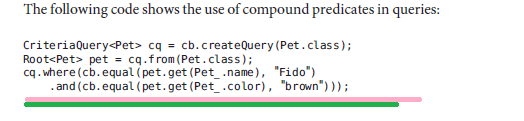


* 1. CriteriaBuilder.between method:



* 1. CriteriaBuilder.like method:

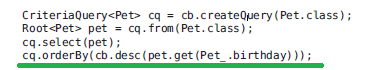


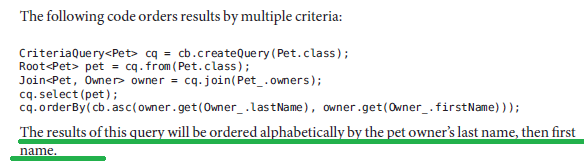
1. 
2. Example
3. 

**Managing CriteriaQuery Results**

1. CriteriaQuery interface defines the **orderBy** method to order query results according to attributes of an entity. The CriteriaQuery interface also defines the **groupBy** method to group the results of a query together according to attributes of an entity, and the having method to restrict those groups according to a condition.

**Ordering Results**

1. The order of the results of a query can be set by calling the **CriteriaQuery.orderBy** **method and passing in an Order object.** **Order objects are created by calling either the CriteriaBuilder.asc or the CriteriaBuilder.desc method**.
2. 
3. If more than one Order object is passed to orderBy, the precedence is determined by the order in which they appear in the argument list of orderBy. The first Order object has precedence.
4. Example

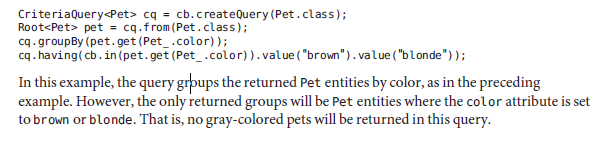


**Grouping Results**

1. The **CriteriaQuery.groupBy** method partitions the query results into groups. These groups are set by passing an expression to groupBy:
2. Example🡺 This query returns all Pet entities and groups the results by the pet’s color



1. The **CriteriaQuery.having** method is used in conjunction with groupBy to filter over the groups
2. Example🡺



**Executing Queries**

1. To prepare a query for execution, create a TypedQuery<T> object with the type of the query result by passing the CriteriaQuery object to EntityManager.createQuery.
2. Queries are executed by calling either **getSingleResult** or **getResultList** **on the TypedQuery<T> object.**