Java Persistence/Criteria

< [Java Persistence](https://en.wikibooks.org/wiki/Java_Persistence)

[Jump to navigation](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#mw-head)[Jump to search](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#p-search)



**Contents**

* [1Criteria API](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Criteria_API)
  + [1.1CriteriaBuilder](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaBuilder)
  + [1.2CriteriaQuery](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaQuery)
    - [1.2.1*CriteriaQuery examples*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaQuery_examples)
    - [1.2.2 **Selection**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Selection)
      * [1.2.2.1 **Aggregation functions**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Aggregation_functions)
      * [1.2.2.2 **Constructors**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Constructors)
    - [1.2.3From](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#From)
      * [1.2.3.1 **Join**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Join)
      * [1.2.3.2 **Fetch**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Fetch)
      * [1.2.3.3 **JoinType**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#JoinType)
    - [1.2.4 **Order**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Order)
    - [1.2.5 **Group By**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Group_By)
    - [1.2.6 **Having**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Having)
  + [1.3CriteriaUpdate (JPA 2.1)](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaUpdate_(JPA_2.1))
    - [1.3.1*CriteriaUpdate examples*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaUpdate_examples)
  + [1.4Criteria**Delete** (JPA 2.1)](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaDelete_(JPA_2.1))
    - [1.4.1*CriteriaDelete examples*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaDelete_examples)
  + [1.5 **Where**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Where)
    - [1.5.1Comparison operations defined on CriteriaBuilder](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Comparison_operations_defined_on_CriteriaBuilder)
    - [1.5.2Logical operations defined on CriteriaBuilder](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Logical_operations_defined_on_CriteriaBuilder)
  + [1.6 **SubQuery**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#SubQuery)
    - [1.6.1*subQuery examples*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#subQuery_examples)
  + [1.7 **Parameters**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Parameters)
    - [1.7.1*Named parameter criteria example*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Named_parameter_criteria_example)
    - [1.7.2*Positional parameter criteria example*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Positional_parameter_criteria_example)
  + [1.8 **Functions**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Functions)
    - [1.8.1CriteriaBuilder database functions](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#CriteriaBuilder_database_functions)
  + [1.9 **Special Operations**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Special_Operations)
    - [1.9.1Criteria API special functions](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Criteria_API_special_functions)
  + [1.10 **Metamodel**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Metamodel)
    - [1.10.1*Metamodel criteria example*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Metamodel_criteria_example)
  + [1.11 **Tuple Queries**](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Tuple_Queries)
    - [1.11.1*Tuple query examples*](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Tuple_query_examples)

Criteria API

API-интерфейс Java Persistence Criteria используется для определения динамических запросов посредством создания объектов определения запросов на основе объектов, а не с использованием строкового подхода JPQL. API критериев позволяет создавать динамические запросы программно, предлагая лучшую интеграцию с языком Java, чем подход 4-й GL на основе строк.

Criteria API имеет два режима: режим с ограничением по типу и режим без ввода. Режим с ограниченным типом использует набор сгенерированного класса мета-модели JPA для определения атрибутов класса с возможностью запроса, см. [Метамодель](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Metamodel) . Нетипизированный режим использует строки для ссылки на атрибуты класса.

API критериев предназначен только для динамических запросов и не может использоваться в метаданных или именованных запросах. Критериальные запросы - это динамические запросы, поэтому они не так эффективны, как статические именованные запросы или даже динамически параметризованные JPQL, которые могут выиграть от использования кеша разбора в некоторых поставщиках JPA.

API Критерии был добавлен в JPA 2.0.

Поддержка удаления и обновления Criteria API была добавлена ​​в JPA 2.1.

CriteriaBuilder

CriteriaBuilder является основным интерфейсом API Criteria. CriteriaBuilder получают из EntityManager или EntityManagerFactory с использованием getCriteriaBuilder () API. CriteriaBuilder используется для создания объектов CriteriaQuery и их выражений. Criteria API в настоящее время поддерживает только запросы выбора.

CriteriaBuilder определяет API для создания объектов CriteriaQuery :

* createQuery () - Создает CriteriaQuery .
* createQuery (Class) - создает CriteriaQuery с использованием обобщений, чтобы избежать приведения класса результата.
* createTupleQuery () - Создает CriteriaQuery, который возвращает карту как объекты Tuple вместо массивов объектов для запросов с несколькими выборками. Смотрите [Tuple Queries](https://en.wikibooks.org/wiki/Java_Persistence/Criteria#Tuple_Queries)
* createCriteriaDelete (Class) - создает CriteriaDelete для удаления пакета объектов непосредственно в базе данных (JPA 2.1).
* createCriteriaUpdate (Class) - создает CriteriaUpdate для обновления пакета объектов непосредственно в базе данных (JPA 2.1).

CriteriaBuilder также определяет все поддерживаемые операции сравнения и функции, используемые для определения предложений запроса.

CriteriaQuery

CriteriaQuery определяет запрос выбора базы данных. А CriteriaQuery все модели из пунктов запроса на выборку JPQL. Элементы из одного CriteriaQuery нельзя использовать в других CriteriaQuerys . CriteriaQuery используется с EntityManager CreateQuery () API для создания JPA запроса .

CriteriaQuery определяет следующие пункты и опции:

* **Different** (boolean) - Определяет, должен ли запрос фильтровать повторяющиеся результаты (по умолчанию false). Если используется соединение с коллекцией отношений, следует использовать различные, чтобы избежать дублирования результатов.
* **from** (Class) – используется для создания root таблицы или итаблиц (если multiselect).
* **from** (EntityType) - Определяет и возвращает элемент в предложении from запроса для типа сущности метамодели. Для правильности запроса требуется хотя бы один элемент from.
* **select (Selection)** - Определяет предложение select для запроса. **Если не установлен, первый “root” будет выбран по умолчанию.**
* **multiselect**(Selection...), multiselect(List<Selection>) - Определяет множественный выбор запроса (выбирает отдельные поля таблицы/таблиц).
* **where**(Expression), where(Predicate...) - Определяет условие where запроса. По умолчанию все экземпляры класса выбраны.
* **orderBy** (Order ...) , orderBy (List <Order>) - Определяет условие заказа запроса. По умолчанию результаты не упорядочены.
* **groupBy** (Expression ...) , groupBy (List <Expression>) - Определяет группу запроса по выражению. По умолчанию результаты не группируются.
* **having**(Expression), having(Predicate...) - Определяет условие запроса. После того, как сгруппированные результаты будут отфильтрованы.
* **subQuery** (Class) - Создает subQuery для использования в одном из других предложений.

Expressions, Predicates, Order элементы определяются с помощью CriteriaBuilder API и выражений , полученный из from Root элементов.

***CriteriaQuery examples***

EntityManagerFactory emf = Persistence.createEntityManagerfactory(“ru.javarush”);

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

CriteriaBuilder cb = em.getCriteriaBuilder();

***//Запросы объектов целиком***

***// Запрос списка объектов.***

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.where(cb.greaterThan(или gt)(root.get("salary"), 100000)); // gt означает - больше чем

//я так понимаю, что это эквивалентно записи

cr.select(root).where(cb.greaterThan(или gt)(root.get("salary"), 100000))

Query query = entityManager.createQuery(cr);

List<Employee> result = query.getResultList();

***// Запрос для одного объекта.***

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.where(cb.equal(root.get("id"), cb.parameter(Long.class, "id")));

//я так понимаю, что это эквивалентно записи

cr.select(root).where(cb.equal(root.get("id"), cb.parameter(Long.class, "id")));

Query query = entityManager.createQuery(cr);

query.setParameter("id", id);

Employee result2 = (Employee)query.getSingleResult();

***//Запросы отдельных переменных объектов***

***// Запрос для одной переменной.***

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.select(cb.max(root.get("salary")));

Query query = entityManager.createQuery(cr);

BigDecimal result3 = (BigDecimal)query.getSingleResult();

***// Запрос списка переменных.***

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.select(root.get("firstName"));

Query query = entityManager.createQuery(cr);

List<String> result4 = query.getResultList();

***// Запрос списка массивов переменных.***

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.multiselect(root.get("firstName"), root.get("lastName"));

Query query = entityManager.createQuery(cr);

List<Object[]> result5 = query.getResultList();

**Selection**

Selection определяет , что выбрано с помощью запроса. Selection может быть любое выражение объекта, выражение атрибута, функция, суб-выбор, конструктор или агрегация функции. alias может быть определен для выбора , используя alias() API.

**Агрегирующие функции**

Функции агрегирования могут включать в себя сводную информацию о наборе объектов. Эти функции могут использоваться для возврата одного результата или с помощью groupBy для возврата нескольких результатов.

Агрегатные функции определены в CriteriaBuilder и включают в себя:

* max (Expression) - возвращает максимальное значение для всех результатов. Используется для числовых типов.
* great (Expression) - Возвращает максимальное значение для всех результатов. Используется для нечисловых типов.
* min (Expression) - возвращает минимальное значение для всех результатов. Используется для числовых типов.
* наименьшее (Expression) - возвращает минимальное значение для всех результатов. Используется для нечисловых типов.
* avg (Expression) - возвращает среднее значение всех результатов. Двойной возвращается.
* sum (Expression) - Возвращает сумму всех результатов.
* sumAsLong (Expression) - возвращает сумму всех результатов. Long возвращается.
* sumAsDouble (Expression) - Возвращает сумму всех результатов. Double возвращается.
* count (Expression) - Возвращает счетчик всех результатов. null значения не учитываются. Long возвращается.
* countDistinct (Expression) - Возвращает количество всех отдельных результатов. нулевые значения не учитываются. Long возвращается.

*// Подсчитать общее количество сотрудников*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.select(cb.count(root));

Query query = entityManager.createQuery(cr);

Long result = query.getSingleResult();

*// Максимальная зарплата*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.select(cb.max(root.get("salary"));

Query query = entityManager.createQuery(cr);

Long result = query.getSingleResult();

**Constructors**

Оператор construct в CriteriaBuilder может использоваться с классом и значениями для возврата объектов данных из запроса критериев. Это не будут управляемые объекты, и класс должен определить конструктор, который соответствует аргументам и типам. Запросы конструктора можно использовать для выбора частичных данных или отчетов по объектам, а также для получения экземпляра класса вместо массива или кортежа объектов.

Т.е. на выходе мы получаем объект другого класса (в нашем случаи это EmpReport) заполненный элементами из “root” класса. При чем, как я понял, этот класс с конструктором должен быть создан зараннее.

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.select(cb.construct(EmpReport.class, root.get("firstName"), root.get("lastName"), root.get("salary")));

Query query = entityManager.createQuery(cr);

List<EmpReport> result = query.getResultList();

**From**

Запрос *from* предложения определяет, что запрашивается. Предложение *from* определяется с помощью API from в CriteriaQuery . Root объект является возвращение из from , которые представляют объект в контексте запроса. Root также From, и Path. From определяет переменную в предложении *from* и разрешает объединения. Path определяет любое значение атрибута и позволяет переходить к вложенным атрибутам.

Root root = cr.from(Employee.class);

Запросы по критериям позволяют использовать несколько объектов корневого уровня. При этом следует соблюдать осторожность, так как это может привести к декартовым произведениям из двух таблиц. Предложение *where* должно обеспечить соединение двух объектов.

*// Выберет сотрудников и почтовые адреса с одинаковым адресом.*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

Root address = cr.from(MailingAddress.class);

cr.multiselect(root, address);

cr.where(cb.equal(root.get("address"), address.get("address"));

Query query = entityManager.createQuery(cr);

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

**Join**

Операция join может использоваться на объекте From для получения отношения, используемого в запросе. join не означает, что отношения будут извлечены, для извлечения связанных объектов в результате используйте вместо этого операцию fetch.

Root root = cr.from(Employee.class);

Join address = root.join("address");

cr.where(cb.equal(address.get("city"), city);

The join operation can be used with OneToOne, ManyToOne, OneToMany, ManyToMany and ElementCollection mappings. При использовании с отношением коллекции вы можете присоединиться к одному и тому же отношению несколько раз, чтобы запросить несколько независимых значений.

*// Все сотрудники, которые работают над обоими проектами.*

Root root = cr.from(Employee.class);

Join p1 = root.join("projects");

Join p2 = root.join("projects");

cr.where(cb.and(cb.equal(p1.get("name"), projectName1), cb.equal(p2.get("name"), projectName2));

**Fetch (это Join Fetch для JPQL)**

**При таком запросе все данные из основной “root” таблицы и связанных таблиц будут запрошены одним SQL запросом.**

Операция fetch может использоваться для объекта From для извлечения связанных объектов в одном запросе. Это позволяет избежать дополнительных запросов для каждого из отношений объекта и гарантирует, что отношения были выбраны, если они были LAZY.

Root root = cr.from(Employee.class);

Fetch address = root.fetch("address");

cr.select(root);

Следует использовать осторожность при использовании Fetch в предложении с where, поскольку это может повлиять на данные, возвращаемые для отношений результирующего объекта. Объекты обычно должны всегда иметь одинаковые данные, независимо от того, как они были запрошены, это важно для кэширования и согласованности. Это проблема, только если псевдоним используется в предложении с where отношения коллекции для фильтрации связанных объектов, которые будут выбраны. Так делать не нужно, но иногда приходится, и в этом случае запрос должен убедиться, что он установлен на BYPASS кеш.

**JoinType**

По умолчанию join и fetch являются INNER joins. Это означает, что результаты, которые не имеют отношения *(т.е. если в ячейке другого сопоставленного столба нет данных или наоборот)*, будут отфильтрованы из результатов запроса. Чтобы избежать этого, объединение может быть определено как OUTER join с использованием LEFT JoinType в качестве аргумента операции **join** или **fetch**.

Root root = cr.from(Employee.class);

Join address = root.join("address", JoinType.LEFT);

cr.order(address.get("city"));

**Примеры Join**

@Entity

public class **Address** {

@Id @GeneratedValue

private Long id;

private String street;

private String city;

private String zipCode;

@ManyToOne

private User owner;

}

@Entity

public class **User** {

@Id @GeneratedValue

private Long id;

private String firstName;

private String lastName;

private int credits;

private Login login;

@OneToMany(cascade = CascadeType.PERSIST, mappedBy = "owner")

private Set<Address> addresses = new HashSet<Address>();

@Temporal(TemporalType.DATE)

private Date birthDate;

@Enumerated(EnumType.STRING)

private Gender gender;

}

@Transactional

public class UserManager {

//get all users living in Paris

@Transactional

public List<User> getUsersFromParis() {

EntityManager em = lazyEM.get();

CriteriaBuilder cb = em.getCriteriaBuilder();

final CriteriaQuery<**User**> cr = cb.createQuery( User.class );

final Root<**User**> root = cr.from( User.class );

final SetJoin<User,Address> a = root.join(User\_.addresses ); // мы получили объект “a” из которого

// уже будем выбирать поля сущности “Address”

root.fetch(User\_.addresses ); //это доп.настройка, говорящая, что поля разных сущностей

// должны “выгребаться” одним запросом

cr.select(root).distinct( true ).where( cb.equal( a.get( Address\_.city ), "Paris" ) );

final TypedQuery<User> typedQuery = em.createQuery( cr );

typedQuery.setFirstResult( 0 ).setMaxResults( 20 );

final List<User> resultList = typedQuery.getResultList();

return resultList;

}

**Пример с каскадом из 3-х связанных таблиц**

**(получаем столбец из нижней таблицы)**

**public** **static** Specific<JpaAction> hasArtifact(**final** String controllerId, **final** String sha1Hash) {

**return** (actionRoot, query, criteriaBuilder) -> {

Join<Action, DistrSet> **dsJoin** = **root**.join(Action\_.distrnSet);

SetJoin<DistrSet, SoftwareModule> **modulesJoin** = **dsJoin**.join(DistrSet\_.modules);

ListJoin<SoftwareModule, Artifact> **artifactsJoin** = **modulesJoin**.[join](https://www.codota.com/code/java/methods/javax.persistence.criteria.SetJoin/join)(SoftwareModule\_.artifacts);

**return** criteriaBuilder.and

(criteriaBuilder.equal(**artifactsJoin**.get(JpaArtifact\_.sha1Hash), sha1Hash),

criteriaBuilder.equal(**root**.get(JpaAction\_.target).get(JpaTarget\_.controllerId),

controllerId));

};

}

**Order**[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=12" \o "Edit section: Order)]

The query *order by* clause defines how the query results will be ordered. The *order by* clause is defined using the orderBy API on CriteriaQuery. Only Order objects can be passed to orderBy, and are obtained from CriteriaBuilder using the asc or desc API.

*// Order by the last and first names.*

Root root = cr.from(Employee.class);

cr.orderBy(cb.desc(root.get("lastName")), cb.asc(root.get("firstName")));

*// Order by the last name, ignoring case.*

Root root = cr.from(Employee.class);

cr.orderBy(cb.asc(cb.upper(root.get("lastName"))));

*// Order by the address object (orders by its id).*

Root root = cr.from(Employee.class);

cr.orderBy(cb.asc(root.get("address")));

**Group By**[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=13" \o "Edit section: Group By)]

The query *group by* clause allows for summary information to be computed on a set of objects. *group by* is normally used in conjunction with aggregation functions. The *group by* clause is defined using the groupBy API on CriteriaQuery with any valid Expression object.

*// Select the average salaries grouped by city.*

Root root = cr.from(Employee.class);

cr.multiselect(cb.avg(root.<Number>get("salary")), root.get("address").get("city"));

cr.groupBy(root.get("address").get("city"));

*// Select the average salaries grouped by city, ordered by the average salary.*

Root root = cr.from(Employee.class);

Expression avg = cb.avg(root.<Number>get("salary"));

cr.multiselect(avg, root.get("address").get("city"));

cr.groupBy(root.get("address").get("city"));

cr.orderBy(cb.asc(avg));

*// Select roots and the count of their number of projects.*

Root root = cr.from(Employee.class);

Expression project = root.join("projects", JoinType.LEFT);

cr.multiselect(root, cb.count(project));

cr.groupBy(root);

**Having** **(это как Where только может применяется после “groupBy”)**

The query *having* clause allows for the results of a *group by* to be filtered. The *having* clause is defined using the having API on CriteriaQuery with any Predicate object.

*// Select the average salaries grouped by city, only including cities with average salaries over 100000.*

Root root = cr.from(Employee.class);

Expression avg = cb.avg(root.<Number>get("salary"));

cr.multiselect(avg, root.get("address").get("city"));

cr.groupBy(root.get("address").get("city"));

cr.having(cb.greaterThan(avg, 100000));

CriteriaUpdate (JPA 2.1)[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=15" \o "Edit section: CriteriaUpdate (JPA 2.1))]

CriteriaUpdate defines a database update query. A CriteriaUpdate models all of the clauses of a JPQL update query. Elements from one CriteriaUpdate cannot be used in other CriteriaUpdates. A CriteriaUpdate is used with the EntityManager createQuery() API to create a JPA Query. Criteria updates should only be used for batch updates. For regular updates to objects, the objects should be read through the EntityManager in a transaction an modified in Java, and have the changes committed.

CriteriaUpdate defines the following clauses and options:

* set(String, Object), set(Path, Object), set(Path, Expression) - Defines the update's *set* clause.
* where(Expression), where(Predicate...) - Defines the update's *where* clause. By default all instances of the class are updated.

The Expressions, Predicates elements are defined using the CriteriaBuilder API and expressions derived from the from Root elements.

***CriteriaUpdate examples***[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=16" \o "Edit section: CriteriaUpdate examples)]

*// Обновляет зарплату до 90 000 из всех сотрудников, делающих более 100 000.*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaUpdate update = cb.createCriteriaUpdate(Employee.class);

Root root = update.from(Employee.class);

update.set("salary", 90000);

update.where(cb.greaterThan(root.get("salary"), 100000));

Query query = entityManager.createQuery(update);

int rowCount = query.executeUpdate();

*// Дает всем сотрудникам повышение на 10%.*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaUpdate update = cb.createCriteriaUpdate(Employee.class);

Root root = update.from(Employee.class);

update.set(root.get("salary"), cb.sum(root.get("salary"),

cb.quot(root.get("salary"), 10));

Query query = entityManager.createQuery(update);

int rowCount = query.executeUpdate();

CriteriaDelete (JPA 2.1)[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=17" \o "Edit section: CriteriaDelete (JPA 2.1))]

CriteriaDelete defines a database delete query. A CriteriaDelete models all of the clauses of a JPQL delete query. Elements from one CriteriaDelete cannot be used in other CriteriaDelete. A CriteriaDelete is used with the EntityManager createQuery() API to create a JPA Query. CriteriaDelete should only be used for batch deletes. For regular deletes of objects, the objects should be read through the EntityManager in a transaction and deleted through the remove() API, and have the changes committed.

CriteriaDelete defines the following clauses and options:

* where(Expression), where(Predicate...) - Defines the delete's *where* clause. By default all instances of the class are deleted.

The Expressions, Predicates elements are defined using the CriteriaBuilder API and expressions derived from the from Root elements.

***CriteriaDelete examples***[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=18" \o "Edit section: CriteriaDelete examples)]

*// Удаляет всех сотрудников, получающих более 100 000.*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaDelete delete = cb.createCriteriaDelete(Employee.class);

Root root = delete.from(Employee.class);

delete.where(cb.greaterThan(root.get("salary"), 100000));

Query query = entityManager.createQuery(delete);

int rowCount = query.executeUpdate();

*// Удаляет всех сотрудников.*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaDelete delete = cb.createCriteriaDelete(Employee.class);

Root root = delete.from(Employee.class);

Query query = entityManager.createQuery(delete);

int rowCount = query.executeUpdate();

Where[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=19" \o "Edit section: Where)]

The *where* clause is normally the main part of the query as it defines the conditions (predicates) that filter what is returned. The *where* clause is defined using the where API on CriteriaQuery with any Predicate objects. A Predicate is obtained using a comparison operation, or a logical operation on CriteriaBuilder. The isNull, isNotNull, and in operations can also be called on Expression. The not operation can also be called on Predicate

**Операции сравнения, определенные в CriteriaBuilder**[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=20)]

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| equal, notEqual | равный | cb.equal(root.get("firstName"), "Bob") |
| lessThan, lt | меньше, чем | cb.lessThan(root.get("salary"), 100000) |
| greaterThan, gt | больше чем | cb.greaterThan(root.get("salary"), cb.parameter(Integer.class, "sal")) |
| lessThanOrEqualTo, le | меньше или равно | cb.lessThanOrEqualTo(root.get("salary"), 100000) |
| greaterThanOrEqualTo, ge | больше или равно | cb.greaterThanOrEqualTo(root.get("salary"), cb.parameter(Integer.class, "sal")) |
| like, notLike | оценивает, являются ли две строки совпадения, '%' и '\_' допустимыми подстановочными знаками, и ESCAPE-символ необязателен | cb.like(root.get("firstName"), "A%")  cb.notLike(root.get("firstName"), "%.\_%", '.') |
| between | оценивает, находится ли значение между двумя значениями | cb.between(root.<String>get("firstName"), "A", "C") |
| isNull | сравнивает значение со значением null, базы данных могут не разрешать или иметь неожиданные результаты при использовании = с нулем | cb.isNull(root.get("endDate"))  root.get("endDate").isNull() |
| in | оценивает, содержится ли значение в списке | cb.in(root.get("firstName")).value("Bob").value("Fred").value("Joe")  root.get("firstName").in("Bob", "Fred", "Joe")  root.get("firstName").in(cb.parameter(List.class, "names") |

**Логические операции, определенные в CriteriaBuilder**[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=21)]

|  |  |  |
| --- | --- | --- |
| **Operation** | **Description** | **Example** |
| and | два или более предикатов вместе | cb.and(cb.equal(root.get("firstName"), "Bob"), cb.equal(root.get("lastName"), "Smith")) |
| or | *or* two or more predicates together | cb.or(cb.equal(root.get("firstName"), "Bob"), cb.equal(root.get("firstName"), "Bobby")) |
| not | negate a predicate | cb.not(cb.or(cb.equal(root.get("firstName"), "Bob"), cb.equal(root.get("firstName"), "Bobby")))  cb.or(cb.equal(root.get("firstName"), "Bob"), cb.equal(root.get("firstName"), "Bobby")).not() |
| conjunction | predicate for true | Predicate where = cb.conjunction();  **if** (name != **null**) {  where = cb.and(where, cb.equal(root.get("firstName"), name));  } |
| disjunction | predicate for false | Predicate where = cb.disjunction();  **if** (name != **null**) {  where = cb.or(where, cb.equal(root.get("firstName"), name));  } |

SubQuery[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=22" \o "Edit section: SubQuery)] (читаются подзапросы очень тяжело)

Подзапросы могут использоваться только в where предложении и having предложении. A subQuery is created from a CriteriaQuery using the subQuery operation. В большинстве случаев использование подзапроса ограничивает подзапрос возвратом одного результата и значения, исключением является использование with the CriteriaBuilder exists, all, any, or some operations, or with an in operation.

***subQuery examples***[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=23)]

*// Найти сотрудника с самой низкой зарплатой.*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

Subquery subQuery = cr.subquery(Employee.class);

Root root\_2 = subQuery.from(Employee.class);

subQuery.select(root\_2.get("salary"));

cr.where(cb.lessThan(root.get("salary"), cb.all(subQuery))); //т.е. значение "salary" должно быть меньше или равно всех (all) значений подзапроса subQuery

Query query = entityManager.createQuery(cr)

List<Employee> = query.getResultList();

*// Найти всех менеджеров, которые управляют только сотрудниками ниже среднего.*

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

Subquery subQuery = cr.subquery(Employee.class);

Root root\_2 = subQuery.from(Employee.class);

subQuery.where(cb

.and(

root\_2.get("manager").equal(e)**,**

cb.equal(root\_2.get("productivity"), "below average"

)

.not()

);

cr.where(cb.exists(subQuery).not());

Query query = entityManager.createQuery(cr)

List<Employee> = query.getResultList();

Parameters[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=24" \o "Edit section: Parameters)]

Параметры могут быть определены с помощью parameter API on CriteriaBuilder. JPA определяет:

- именованные параметры, для которых указываются тип и имя параметра,

- позиционные параметры, для которых указывается только тип параметра. Позиционные параметры начинаются с позиции **1**, а не с  **0** .

***Named parameter criteria example***[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=25" \o "Edit section: Named parameter criteria example)]

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.where(

cb.equal(root.get("firstName"), cb.parameter(String.class, "first")),

cb.equal(root.get("lastName"), cb.parameter(String.class, "last"))

);

Query query = entityManager.createQuery(cr)

query.setParameter("first", "Bob");

query.setParameter("last", "Smith");

List<Employee> = query.getResultList();

***Positional parameter criteria example***[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=26" \o "Edit section: Positional parameter criteria example)]

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root root = cr.from(Employee.class);

cr.where(

cb.equal(root.get("firstName"), cb.parameter(String.class)),

cb.equal(root.get("lastName"), cb.parameter(String.class))

);

Query query = entityManager.createQuery(cr)

query.setParameter(1, "Bob");

query.setParameter(2, "Smith");

List<Employee> = query.getResultList();

Functions[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=27" \o "Edit section: Functions)]

Criteria API поддерживает несколько функций базы данных. Все поддерживаемые функции определены в CriteriaBuilder. Некоторые функции могут не поддерживаться некоторыми базами данных, если они не совместимы с SQL и не предлагают эквивалентных функций.

**CriteriaBuilder database functions**[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=28" \o "Edit section: CriteriaBuilder database functions)]

|  |  |  |
| --- | --- | --- |
| **Function** | **Description** | **Example** |
| diff | вычитание | cb.diff(root.<Number>get("salary"), 1000) |
| sum | прибавление | cb.sum(root.<Number>get("salary"), 1000) |
| prod | умножение | cb.prod(root.<Number>get("salary"), 2) |
| quot | деление | cb.quot(root.<Number>get("salary"), 2) |
| abs | абсолютная величина | cb.abs(  cb.diff(root.<Number>get("salary"), root.get("manager").<Number>get("salary"))) |
| selectCase | определяет заявление случая | cb.selectCase(root.get("status")).  when(0, "active").  when(1, "consultant").  otherwise("unknown")  cb.selectCase().  when(cb.equal(root.get("status"), 0), "active").  when(cb.equal(root.get("status"), 1), "consultant").  otherwise("unknown") |
| coalesce | оценивается как первое ненулевое значение аргумента | cb.coalesce(cb.concat(root.<Number>get("salary"), 0) |
| concat | объединяет два или более строковых значения | cb.concat(  cb.concat(root.<String>get("firstName"), " "), root.<String>get("lastName")) |
| currentDate | текущая дата в базе данных | cb.currentDate() |
| currentTime | текущее время в базе данных | cb.currentTime() |
| currentTimestamp | текущая дата-время в базе данных | cb.currentTimestamp() |
| length | длина символа / байта символа или двоичное значение | cb.length(root.<String>get("lastName")) |
| locate | индекс строки в строке, необязательно начиная с начального индекса | cb.locate("-", root.<String>get("lastName")) |
| lower | преобразовать строковое значение в нижний регистр | cb.lower(root.<String>get("lastName")) |
| mod | вычисляет остаток от деления первого целого числа на второе | cb.mod(root.<Integer>get("hoursWorked"), 8) |
| nullif | возвращает ноль, если первый аргумент равен второму аргументу, в противном случае возвращает первый аргумент | cb.nullif(root.<Number>get("salary"), 0) |
| sqrt | вычисляет квадратный корень из числа | cb.sqrt(root.<Number>get("salary")) |
| substring | подстрока из строки, начиная с индекса, необязательно с размером подстроки | cb.substring(root.<String>get("lastName"), 0, 2) |
| trim | обрезает начальный, конечный или оба пробела или необязательный символ обрезки строки | cb.trim(TrimSpec.TRAILING, root.<String>get("lastName"))  cb.trim(root.<String>get("lastName"))  cb.trim(TrimSpec.LEADING, '-', root.<String>get("lastName")) |
| upper | преобразовать строковое значение в верхний регистр | cb.upper(root.<String>get("lastName")) |

Special Operations[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=29" \o "Edit section: Special Operations)]

Criteria API определяет несколько специальных операций, которые не являются функциями базы данных, но имеют особое значение в JPA. Некоторые из этих операций определены в CriteriaBuilder, а некоторые - в определенных интерфейсах Expression.

**Criteria API special functions**[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=30" \o "Edit section: Criteria API special functions)]

|  |  |  |
| --- | --- | --- |
| **Function** | **Description** | **Example** |
| index | индекс упорядоченного элемента List, поддерживается только тогда, когда в сопоставлении используется @OrderColumn,  определено в интерфейсе ListJoin, полученном из элемента From с помощью операции joinList | Root root = cr.from(Employee.class);  ListJoin toDo = root.joinList("toDoList");  cr.multiselect(e, toDo);  cr.where(cb.equal(toDo.index(), 1)); |
| key, value | ключ или значение элемента Map, определенного в интерфейсе MapJoin, полученного из элемента From с помощью операции joinMap | Root root = cr.from(Employee.class);  MapJoin p = root.joinMap("priorities");  cr.multiselect(e, p.value());  cr.where(cb.equal(p.key(), "high")) |
| size | размер отношений коллекции, это оценивает к под-выбору, определенному в CriteriaBuilder | cb.greaterThan(cb.size(root.<Collection>get("managedEmployees")), 2) |
| isEmpty, isNotEmpty | оценивается как истина, если отношение коллекции пустое или нет, это оценивается как дополнительный выбор, определенный в CriteriaBuilder | cb.isEmpty(root.<Collection>get("managedEmployees")) |
| isMember, isNotMember | оценивается как истина, если отношение коллекции содержит значение, это оценивается как дополнительный выбор, определенный в CriteriaBuilder | cb.isMember("write code", root.<Collection>get("responsibilities")) |
| type | значение дискриминатора наследования, определенное в любом выражении Path | cb.equal(p.type(), LargeProject.class) |
| as | может использоваться для приведения нетипизированного выражения к типизированному выражению, EclipseLink также позволяет это преобразовывать наследуемые типы вниз | cb.mod(root.get("id").as(Integer.class), 2)  cb.greaterThan(p.as(LargeProject.class).get("budget"), 1000000) |
| function | вызвать специфическую для базы данных функцию, определенную в CriteriaBuilder | cb.greaterThan(cb.function("TO\_NUMBER", Number.class, p.get("areaCode")), 613) |

Metamodel[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=31" \o "Edit section: Metamodel)]

JPA определяет метамодель, которую можно использовать во время выполнения для запроса информации о метаданных отображения ORM. Мета-модель включает в себя список сопоставленных атрибутов для класса, а также их типы сопоставления и количество элементов. Мета-модель может использоваться с Criteria API вместо использования строк для ссылки на атрибуты класса.

JPA определяет набор \_ классов, которые должны быть сгенерированы провайдером JPA или IDE, которые предоставляют доступ во время компиляции к метамодели. Это позволяет использовать типизированные статические переменные в Criteria API. Это может уменьшить количество опечаток или недопустимых запросов в коде приложения, обнаруживая проблемы запросов во время компиляции, а не во время тестирования. Это, однако, добавляет сложности в процесс разработки, так как статический класс метамодели должен быть сгенерирован и быть частью цикла разработки.

***Metamodel criteria example***[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=32" \o "Edit section: Metamodel criteria example)]

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery cr = cb.createQuery();

Root<Employee> root = cr.from(entityManager.getMetamodel().entity(Employee.class));

cr.where(cb.equal(root.get(Employee\_.firstName), "Bob"), cb.equal(root.get(Employee\_.lastName), "Smith"));

Query query = entityManager.createQuery(cr)

List<Employee> = query.getResultList();

Tuple Queries[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=33" \o "Edit section: Tuple Queries)]

**Кортеж** определяет множественный выбор результата запроса (из результатов запроса можно выводить не сразу все элементы, а по отдельности). Обычно массив объектов возвращается запросами множественного выбора JPA, но массив объектов не очень полезная структура данных. **Кортеж** представляет собой карто-подобную структуру, которая позволяет позволяет получать отдельные результаты по имени или индексу.

***Tuple query examples***[[edit](https://en.wikibooks.org/w/index.php?title=Java_Persistence/Criteria&action=edit&section=34" \o "Edit section: Tuple query examples)]

CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Tuple> cr = cb.createTupleQuery();

Root root = cr.from(Employee.class);

cr.multiselect(root.get("firstName").alias("first"), root.get("age").alias("ag"));

Query query = entityManager.createQuery(cr);

List<Tuple> results = query.getResultList();

String first = (String)results.get(0).get("first");

Integer age = (Integer)results.get(0).get("ag");