**🔍 What is DetachedCriteria in Hibernate?**

* DetachedCriteria is a type of **Criteria** that is **not tied to any Hibernate Session** when it is created.
* It allows you to **build query logic independently**, and later **attach it to a Session** to execute the query.
* Especially useful in **multi-layered applications** (like DAO and service layers) where query logic is created at one layer and executed at another.

**🎯 Why Use DetachedCriteria for Joins?**

When using **joins with DetachedCriteria**, it helps to:

1. **Create reusable subqueries** or joins.
2. **Avoid coupling query construction with session access**, making your code more modular and testable.
3. Enable **dynamic and conditional joins**.
4. **Execute queries in a different session context**, helpful for stateless or disconnected applications.

Since you're using **Hibernate 6**, it's important to know that **DetachedCriteria is removed** in Hibernate 6. The Criteria API in Hibernate 6 is now **fully JPA-compliant**, and you should use **JPA Criteria API** instead of Hibernate's proprietary APIs like DetachedCriteria.

**❌ DetachedCriteria in Hibernate 6?**

* **Deprecated in Hibernate 5**
* **Removed in Hibernate 6**
* Use **JPA Criteria API with subqueries and joins** instead.

**✅ What to Use Instead in Hibernate 6?**

Use **CriteriaBuilder** and **CriteriaQuery** with join() or fetch() for joins.

DetachedCriteria was **removed in Hibernate 6** for the following key reasons:

**🔍 1. Non-standard / Hibernate-specific API**

* DetachedCriteria was a **proprietary Hibernate API** (not part of JPA).
* Hibernate 6 is designed to **align more closely with the JPA standard**.
* The goal was to reduce Hibernate-specific APIs and focus on JPA’s **Criteria API**, which is more portable and interoperable.

**🚫 2. Complexity & Maintainability**

* DetachedCriteria added **complexity to the internal query framework**.
* Maintaining two separate query-building APIs (Criteria and DetachedCriteria) made it harder to evolve the internals.
* Hibernate 6 introduces a **completely new query engine**, so keeping DetachedCriteria would require major rework.

**⚙️ 3. Redundancy**

* Almost everything DetachedCriteria could do (joins, filters, subqueries, reusable logic) can now be done with:
  + CriteriaBuilder
  + CriteriaQuery
  + Subquery
  + Reusable utility methods to build Predicate or Expression

**📦 4. Encouraging Better Practices**

* DetachedCriteria encouraged **query construction outside the session**, which may cause problems like:
  + Type safety issues
  + Difficulty in debugging
  + Poor separation of concerns (query logic mixed with UI/service layers)

Hibernate 6 encourages **clear, safe, and session-aware query building**, typically at the repository or DAO level.

**✅ What Should You Use Instead?**

| **Old (DetachedCriteria)** | **New in Hibernate 6** |
| --- | --- |
| DetachedCriteria.forClass() | CriteriaBuilder.createQuery() |
| createAlias() | root.join("relation") |
| add(Restrictions.like()) | cb.like(root.get("property"), value) |
| Reuse criteria logic | Write reusable methods returning Predicate, Subquery, or Expression |

**❓ Was DetachedCriteria Mainly Used for Joins?**

**No, not mainly.**  
DetachedCriteria was not introduced *specifically* for joins. Instead, it was designed to allow:

**✅ Main Purpose of DetachedCriteria**

**To build criteria queries independently of a Hibernate session.**

This allowed the query logic to be:

* Created outside of a session (e.g., in the service layer)
* Reused and executed later in a different session
* Modular and testable

**🔄 Joins with DetachedCriteria**

While DetachedCriteria *supported joins* using createAlias(), that was **just one feature** of it—not the primary purpose.

**✔ You could use it for:**

| **Use Case** | **Supported by DetachedCriteria** |
| --- | --- |
| Joining associations | ✅ createAlias() |
| Adding conditions (filters) | ✅ add(Restrictions...) |
| Projections (select specific columns) | ✅ |
| Sorting | ✅ addOrder(...) |
| Reusability across layers | ✅ Detached from session |

**🔧 Example in Hibernate 5 (Before Hibernate 6)**

DetachedCriteria dc = DetachedCriteria.forClass(Employee.class);

dc.createAlias("department", "dpt");

dc.add(Restrictions.eq("dpt.name", "HR"));

Criteria executableCriteria = dc.getExecutableCriteria(session);

List<Employee> list = executableCriteria.list();

Let’s look at this line again:

Join<Department, Employee> empJoin = deptRoot.join("employees", JoinType.INNER);

**🔍 What is empJoin?**

empJoin is a **reference to the joined Employee table**—it represents the employees **associated with each department** during the join.

**📘 Let's break it down:**

| **Component** | **Meaning** |
| --- | --- |
| Join<Department, Employee> | A **join object** linking Department (root) with Employee (joined entity) |
| deptRoot | The root of your query (main entity: Department) |
| .join("employees") | Joins with the employees collection (from Department entity) |
| "employees" | Refers to the Set<Employee> in Department entity |
| empJoin.get("name") | Refers to the name property of the Employee |

**🔍 CriteriaBuilder Methods**

| **Method** | **Usage Example** | **Purpose** |
| --- | --- | --- |
| equal() | cb.equal(empJoin.get("empname"), "Karan") | Exact match |
| notEqual() | cb.notEqual(empJoin.get("empname"), "Karan") | Not equal |
| greaterThan() | cb.greaterThan(empJoin.get("salary"), 50000) | Greater than (for numbers, dates) |
| lessThan() | cb.lessThan(empJoin.get("salary"), 30000) | Less than |
| greaterThanOrEqualTo() | cb.greaterThanOrEqualTo(empJoin.get("age"), 25) | Greater than or equal |
| lessThanOrEqualTo() | cb.lessThanOrEqualTo(empJoin.get("experience"), 5) | Less than or equal |
| between() | cb.between(empJoin.get("salary"), 40000, 60000) | Between two values |
| in() | empJoin.get("dept").in("HR", "Admin", "Finance") | Value in a list |
| isNull() | cb.isNull(empJoin.get("email")) | Check if null |
| isNotNull() | cb.isNotNull(empJoin.get("email")) | Check if not null |
| and() | cb.and(predicate1, predicate2) | Combine multiple conditions (AND) |
| or() | cb.or(predicate1, predicate2) | Combine multiple conditions (OR) |
| not() | cb.not(cb.equal(...)) | Logical NOT |
| like() with wildcard | cb.like(empJoin.get("name"), "%ar%") | Contains "ar" |
| startsWith() ⇒ via like("X%") | cb.like(empJoin.get("name"), "K%") | Starts with "K" |
| endsWith() ⇒ via like("%X") | cb.like(empJoin.get("name"), "%n") | Ends with "n" |

**🧪 Example using other methods:**

**👉 equal():**

cb.equal(empJoin.get("empname"), "Karan")

**👉 greaterThan():**

cb.greaterThan(empJoin.get("salary"), 50000)

**👉 Combine multiple:**

Predicate p1 = cb.like(empJoin.get("empname"), "K%");

Predicate p2 = cb.greaterThan(empJoin.get("salary"), 40000);

cq.where(cb.and(p1, p2));

**✅ Pro Tip:**

Use **cb.and() / cb.or()** when you have multiple conditions:

cq.where(

cb.and(

cb.like(empJoin.get("empname"), "K%"),

cb.greaterThan(empJoin.get("experience"), 2)

)

);

**🎯 Summary:**

cb.like() is only one of many condition methods. You can use:

* equal, greaterThan, between, in, isNull, not, and, or, etc.