### Derived Query Methods in JPA Repositories

Spring Data JPA provides a powerful mechanism to define query methods based on method names, also known as \*\*Derived Query Methods\*\*. This allows you to query the database without writing any JPQL or SQL by simply following a method-naming convention.

Derived query methods automatically translate the method name into a SQL query, making database interactions more intuitive and less code-intensive.

---

### Structure of Derived Query Methods

The structure of a derived query method follows a specific pattern:

```

findBy[Field][Operation](Parameters)

```

- \*\*`findBy`\*\*: Specifies the beginning of the query method. Other keywords like `readBy`, `queryBy`, or `getBy` can also be used.

- \*\*`[Field]`\*\*: Refers to the entity's field (or property) being queried.

- \*\*`[Operation]`\*\*: Optional operation to modify how the field is matched (e.g., `And`, `Or`, `Like`, `GreaterThan`, `Between`, etc.).

- \*\*`(Parameters)`\*\*: Arguments passed to the method based on the fields being queried.

---

### Examples of Derived Query Methods

Let's assume we have the following `User` entity:

```java

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private int age;

private boolean active;

// Getters and setters

}

```

#### 1. \*\*Simple Queries\*\*

\*\*Find by a single field:\*\*

```java

List<User> findByName(String name);

```

- This query will retrieve all `User` entities where the `name` field matches the given value.

\*\*Find by multiple fields:\*\*

```java

User findByNameAndEmail(String name, String email);

```

- This query will retrieve the `User` entity where both `name` and `email` match the provided values.

\*\*Find by comparison:\*\*

```java

List<User> findByAgeGreaterThan(int age);

```

- This will retrieve all users where `age` is greater than the provided value.

---

#### 2. \*\*Using Logical Operators\*\*

\*\*AND condition:\*\*

```java

List<User> findByNameAndAge(String name, int age);

```

- This query will find users where both `name` and `age` match the given values.

\*\*OR condition:\*\*

```java

List<User> findByNameOrEmail(String name, String email);

```

- This query will find users where either `name` matches the provided name or `email` matches the provided email.

---

#### 3. \*\*Using Comparison Operators\*\*

\*\*Greater than:\*\*

```java

List<User> findByAgeGreaterThan(int age);

```

- Finds users whose age is greater than the specified value.

\*\*Less than:\*\*

```java

List<User> findByAgeLessThan(int age);

```

- Finds users whose age is less than the specified value.

\*\*Between:\*\*

```java

List<User> findByAgeBetween(int startAge, int endAge);

```

- Finds users whose age is between the given start and end values.

---

#### 4. \*\*Using Like and Containing\*\*

\*\*Like for partial matches:\*\*

```java

List<User> findByNameLike(String pattern);

```

- Finds users where `name` matches the provided pattern (e.g., `findByNameLike("%ohn%")` for names like "John").

\*\*Containing for substring matches:\*\*

```java

List<User> findByNameContaining(String substring);

```

- Finds users whose `name` contains the given substring. Equivalent to SQL's `%substring%` search.

\*\*Starting with:\*\*

```java

List<User> findByNameStartingWith(String prefix);

```

- Finds users whose `name` starts with the given prefix.

\*\*Ending with:\*\*

```java

List<User> findByNameEndingWith(String suffix);

```

- Finds users whose `name` ends with the given suffix.

---

#### 5. \*\*Null Checking\*\*

\*\*Find users with null fields:\*\*

```java

List<User> findByEmailIsNull();

```

- Finds users where the `email` field is null.

\*\*Find users with non-null fields:\*\*

```java

List<User> findByEmailIsNotNull();

```

- Finds users where the `email` field is not null.

---

#### 6. \*\*Boolean Fields\*\*

For boolean fields, you can query based on `true` or `false`.

```java

List<User> findByActiveTrue();

List<User> findByActiveFalse();

```

- These queries will return users based on whether the `active` field is `true` or `false`.

---

#### 7. \*\*Order By\*\*

To sort results, you can append `OrderBy` followed by the field name and sorting direction.

```java

List<User> findByAgeOrderByNameAsc(int age);

```

- This will find users of a certain age and sort them by `name` in ascending order.

```java

List<User> findByAgeOrderByNameDesc(int age);

```

- This will sort the users by `name` in descending order.

---

### Query Keywords in Derived Methods

Spring Data JPA supports a variety of keywords that can be used to build complex queries without writing SQL. Some of the most common ones are:

| Keyword | Functionality |

|--------------|---------------|

| `And` | Combines two conditions using `AND`. |

| `Or` | Combines two conditions using `OR`. |

| `Is` | Used for exact matches (e.g., `findByNameIs(String name)` is equivalent to `findByName(String name)`). |

| `Equals` | Checks for equality (same as `Is`). |

| `Between` | Checks if a field's value is between two values. |

| `LessThan` | Checks if a field's value is less than the given value. |

| `GreaterThan`| Checks if a field's value is greater than the given value. |

| `Like` | Searches for a pattern match (SQL `LIKE`). |

| `NotLike` | Searches for non-matching patterns. |

| `In` | Checks if a field's value is in a given list of values. |

| `NotIn` | Checks if a field's value is not in a given list of values. |

| `Containing` | Searches for a substring in a field. |

| `StartingWith` | Checks if a field starts with the given value. |

| `EndingWith` | Checks if a field ends with the given value. |

| `OrderBy` | Orders the result set by the specified field. |

---

### Examples of Derived Query Methods

1. \*\*Find all users by name\*\*:

```java

List<User> findByName(String name);

```

2. \*\*Find users by email or name\*\*:

```java

List<User> findByEmailOrName(String email, String name);

```

3. \*\*Find active users over a certain age\*\*:

```java

List<User> findByActiveTrueAndAgeGreaterThan(int age);

```

4. \*\*Find users by a substring in their name, ordered by age\*\*:

```java

List<User> findByNameContainingOrderByAgeDesc(String substring);

```

5. \*\*Find users with null email\*\*:

```java

List<User> findByEmailIsNull();

```

6. \*\*Find users whose name starts with "Jo" and age between 25 and 35\*\*:

```java

List<User> findByNameStartingWithAndAgeBetween(String prefix, int startAge, int endAge);

```

---

### Conclusion

Derived query methods in Spring Data JPA provide an elegant and declarative way to define database queries simply by writing method signatures. By following naming conventions, you can avoid writing complex SQL or JPQL queries and still perform a wide variety of database operations. This approach helps reduce boilerplate code and improves maintainability, making it one of the most powerful features of Spring Data JPA.