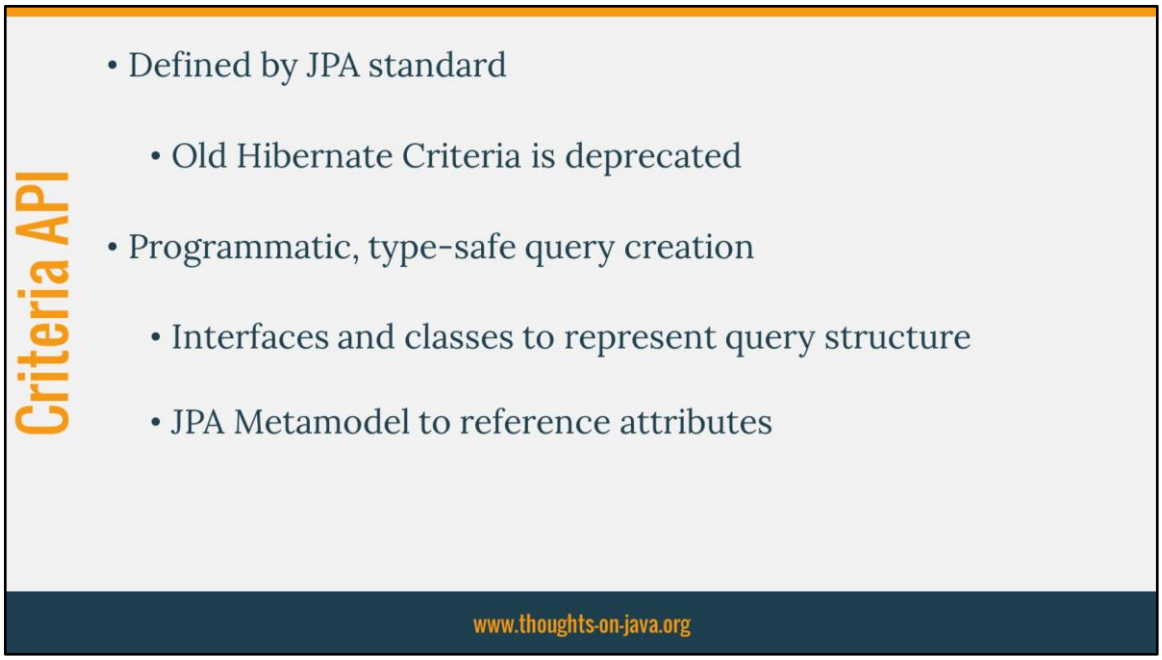
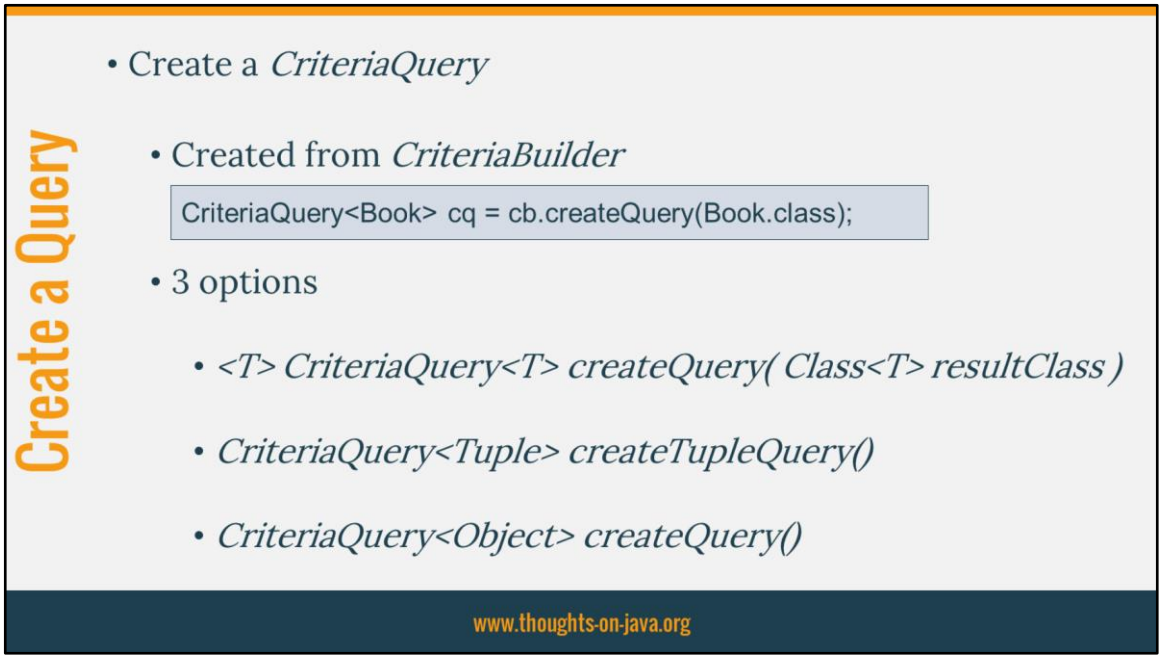
You sometimes need to programmatically define a database query. You can, of course, use basic *String* operations to create a JPQL query and execute it as a dynamic query. But the Criteria API provides a much better solution for these use cases.

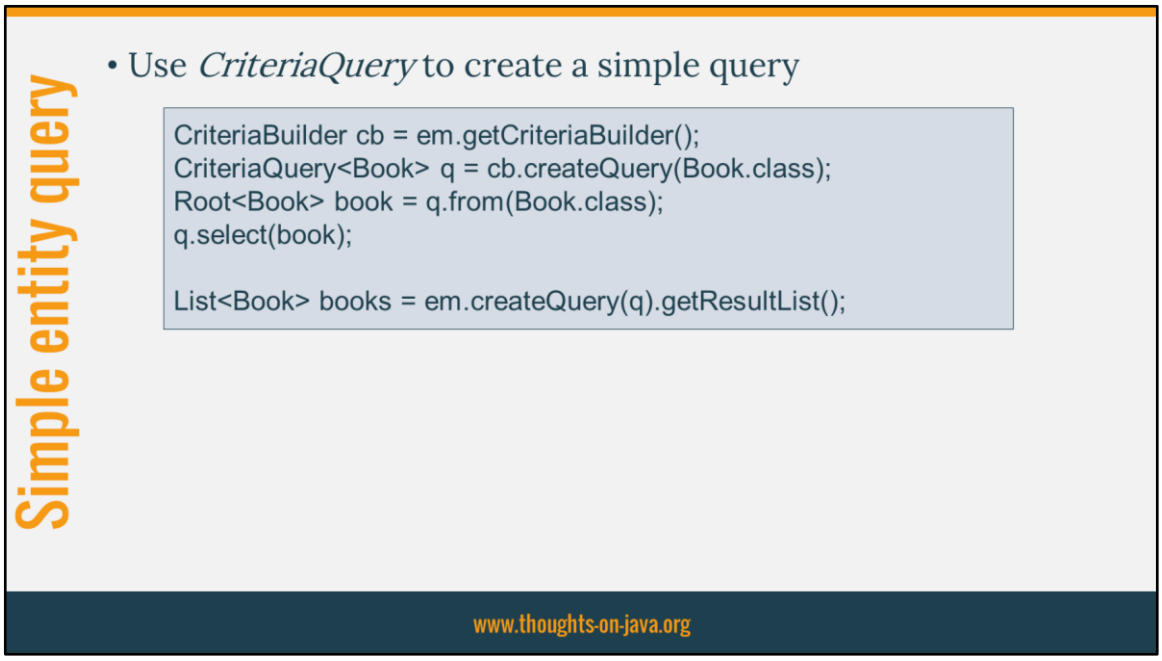


The Criteria API I show you in this video is defined by the JPA standard and replaces Hibernate's proprietary Criteria API. Hibernate's API is deprecated, and the development team recommends to use the new standard API.
The Criteria API provides a programmatic, type-safe way to create queries at runtime. It, therefore, provides a set of interfaces and classes to define the structure of your query. Entity attributes are either referenced by their name or by attributes of JPA Metamodel classes. To keep it easy, I will reference all attributes by their name and explain the JPA Metamodel in the following video. If you want to use the Criteria API in real projects, I recommend to watch both videos and reference entity attributes via the JPA Metamodel.
Let's start with the definition of a simple query and add more complex features along the way.

Before you can create a *CriteriaQuery*, you need to get a *CriteriaBuilder* instance. It acts as a factory to create different parts of the query. You can retrieve an instance of it by calling the *getCriteriaBuilder()* method on the *EntityManager* or the *EntityManagerFactory*.



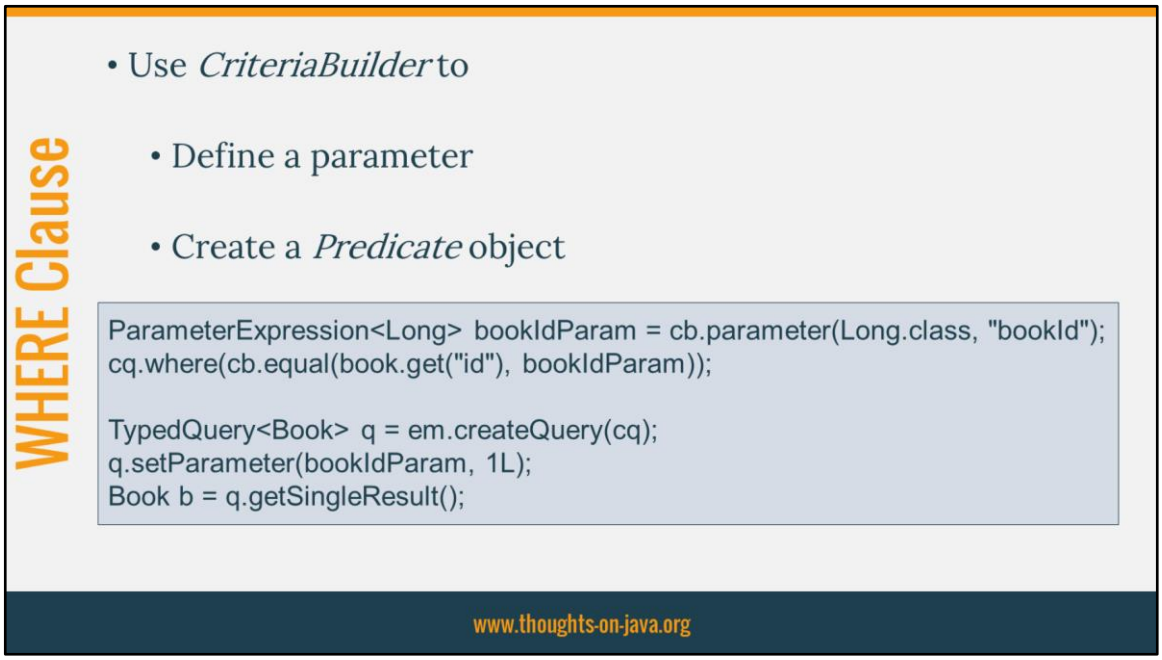
You can then use the *CriteriaBuilder* to create a *CriteriaQuery*. It's the root of the object graph that represents your query.
As you can see here on the slide, you can choose between 3 options to create a *CriteriaQuery*. The first approach is the most common one. It creates a typed *CriteriaQuery* instance and the second is just a convenient way to create a query that selects a *Tuple*. I will get into more details about this kind of query later.



In the next step, you can use the *CriteriaQuery* instance to define your query. The code sample on this slides shows an example of a simple query that selects all *Book* entities from the database.
Let's switch to the IDE and have a more detailed look at it.

Demo

www.thoughts-on-java.org



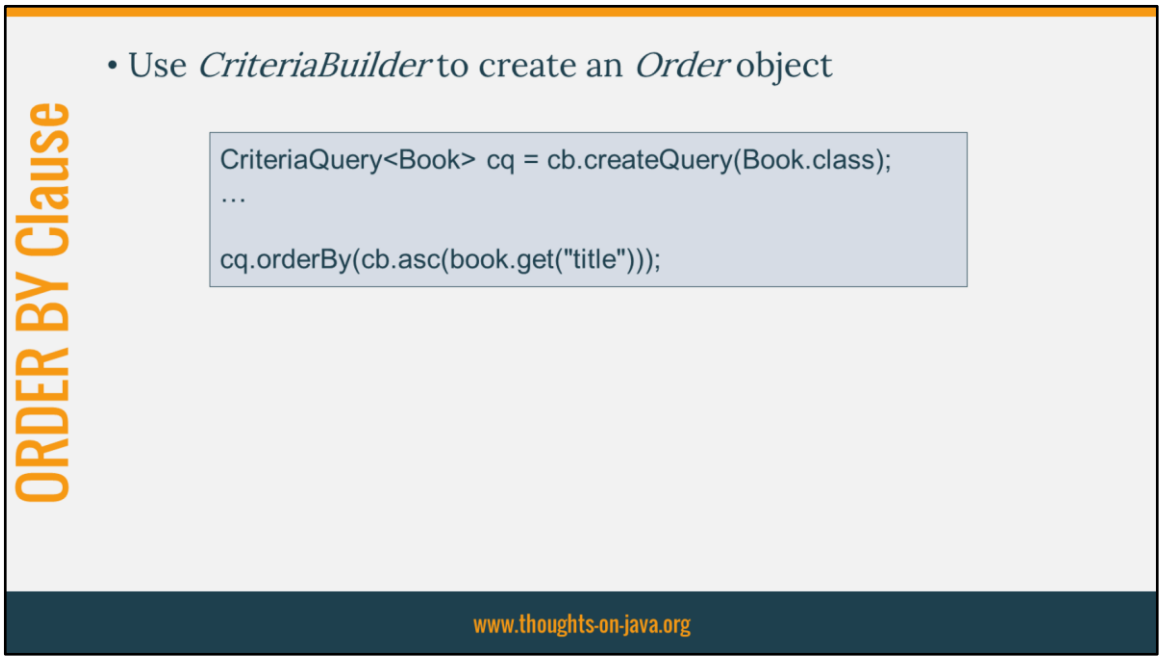
You now know how to use the Criteria API to create a simple query. But you obviously want to do more than just selecting all entities. In the rest of this video, I will show you more features of the Criteria API and how you can use them to create more complex queries.

One thing you will need in almost all queries is the WHERE clause. You can define it by calling the where method on the *CriteriaQuery* interface with a *Predicate*. The *CriteriaBuilder* provides you with several methods to programmatically create *Predicates* of any complexity. But be careful, complex *Predicates* are extremely hard to read. So better add a few comments while you still know what you've done.

The predicate in this example compares the value of the *id* attribute with the value of the named bind parameter *bookId*. The definition of the bind parameter looks different to the definition in a JPQL query. But besides that, there is no difference. You define them by calling the parameter method on the *CriteriaBuilder* and set their value by calling the *setParameter* method on the *Query* interface. Hibernate will then take care of the type handling, escape values if necessary and has the option to apply additional performance optimizations.

Demo

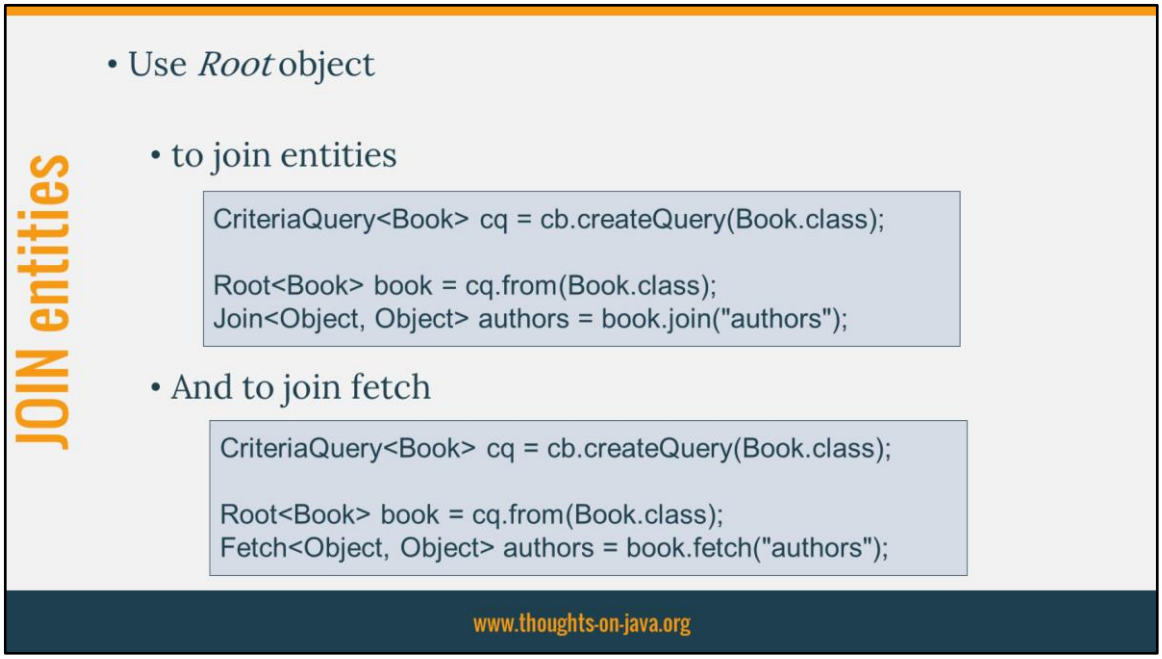
www.thoughts-on-java.org



The definition of the ORDER BY clause is similar to the definition of the WHERE clause. The *CriteriaBuilder* interface provides the *asc* and *desc* method to create an instance of the *Order* interface, and the *orderBy* method of the *CriteriaQuery* adds it to the query.

Demo

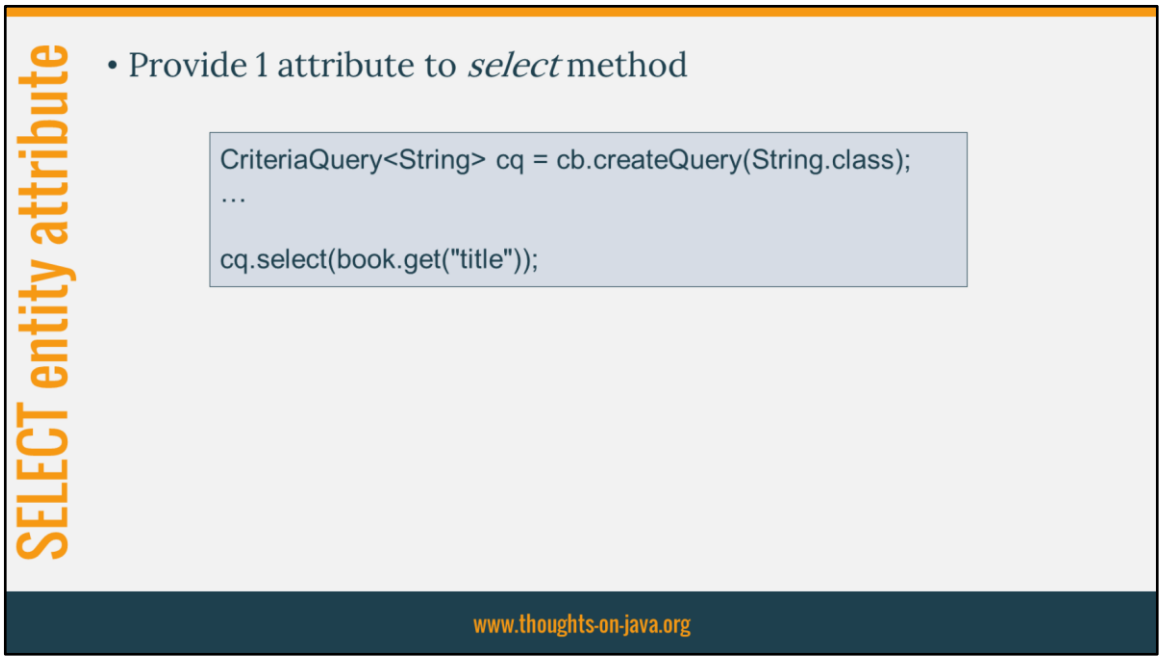
www.thoughts-on-java.org



The next thing I want to show you are JOIN statements. They're used by most queries and easy to define. You just need to call the join method on the *Root* or another *Join* interface and provide the attribute that maps the relationship you want to join. If you want to use a specific join type, like a left outer join, you can provide it as a second parameter.
The definition of a JOIN FETCH statement is pretty similar. You just call the *fetch* instead of the *join* method.

Demo

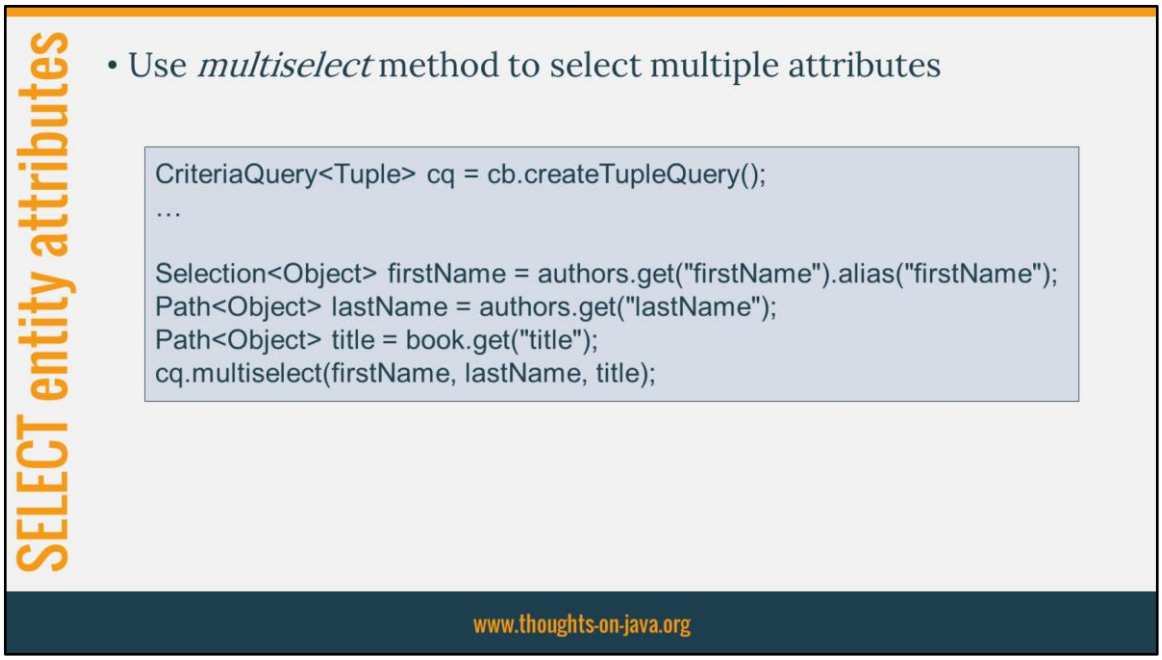
www.thoughts-on-java.org



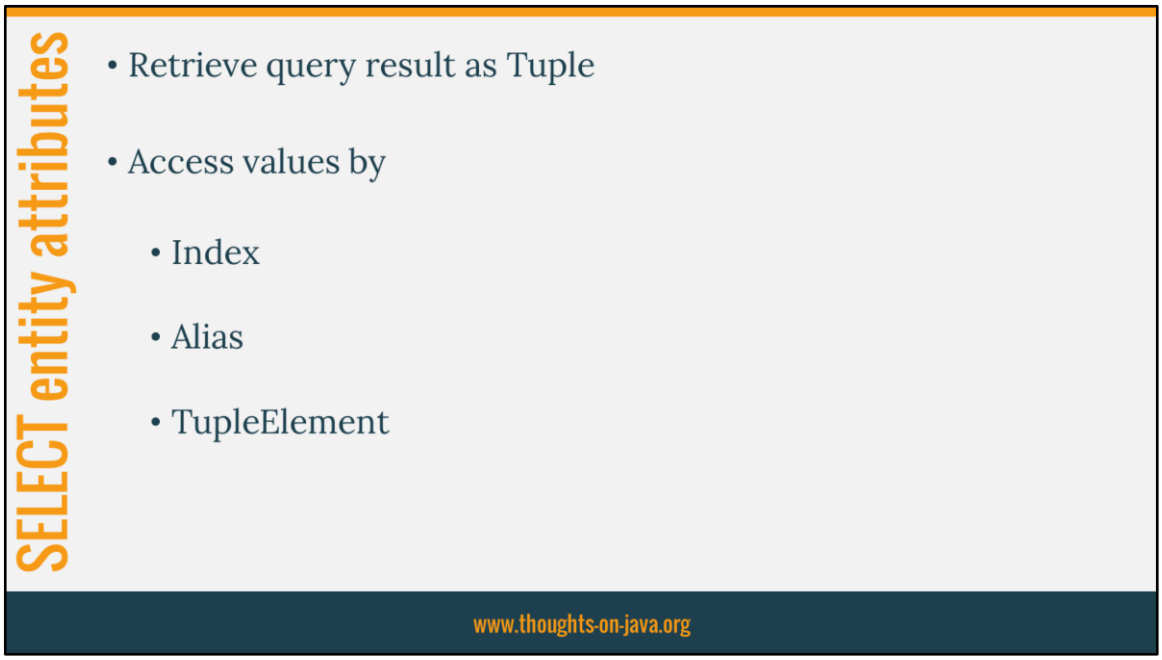
JPQL also supports other projections than just entities. That's the same for the Criteria API. You can select one or more attributes as scalar values or use a constructor expression to map the query result to a constructor call. Each of these projections gets defined in a different way. Let's have a look at the selection of 1 entity attribute first.
As you can see in the code snippet, you just have to reference the entity attribute in the *select* method.

Demo

www.thoughts-on-java.org



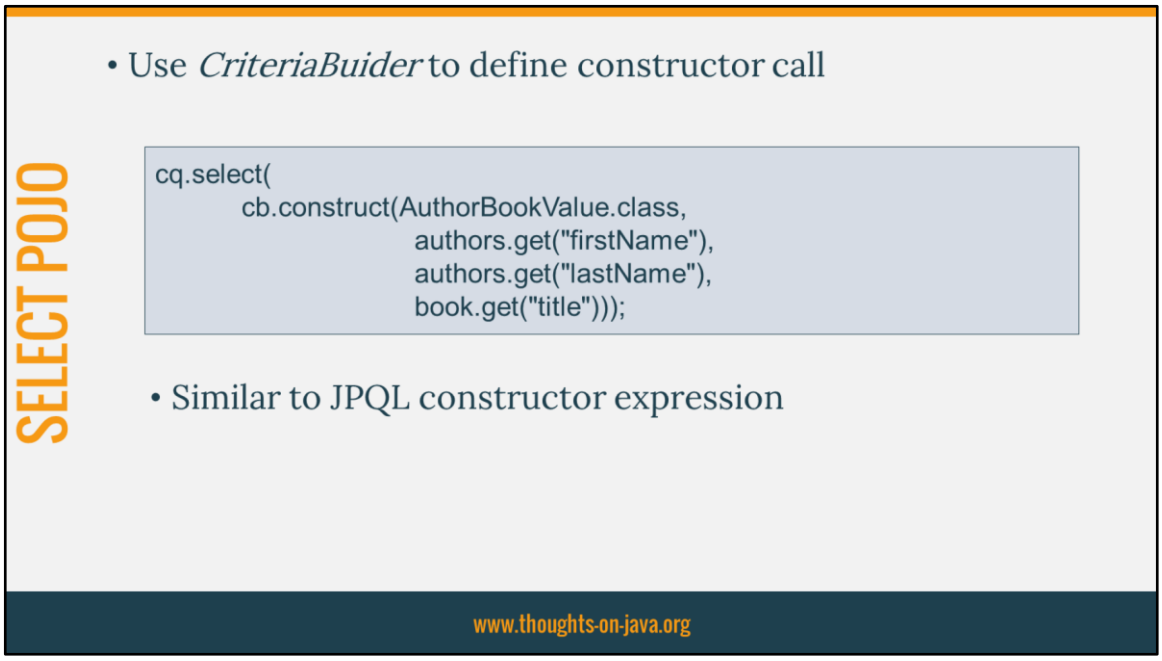
The select method of the *CriteriaQuery* accepts only 1 parameter. You need to use the *multiselect* method if you want to select multiple attributes. In the code snippet on the slide, I use local variables for the *Path* references of the entity attributes. I do that so that I can use them later to retrieve the attributes from the query result. After you execute this query, you can access the selected attributes via the *Tuple* interface.



It allows you easy access to all attributes in the query result via their index, alias or an instance of the *TupleElement* interface. Let's switch to the IDE and have a more detailed look at it.

Demo

www.thoughts-on-java.org

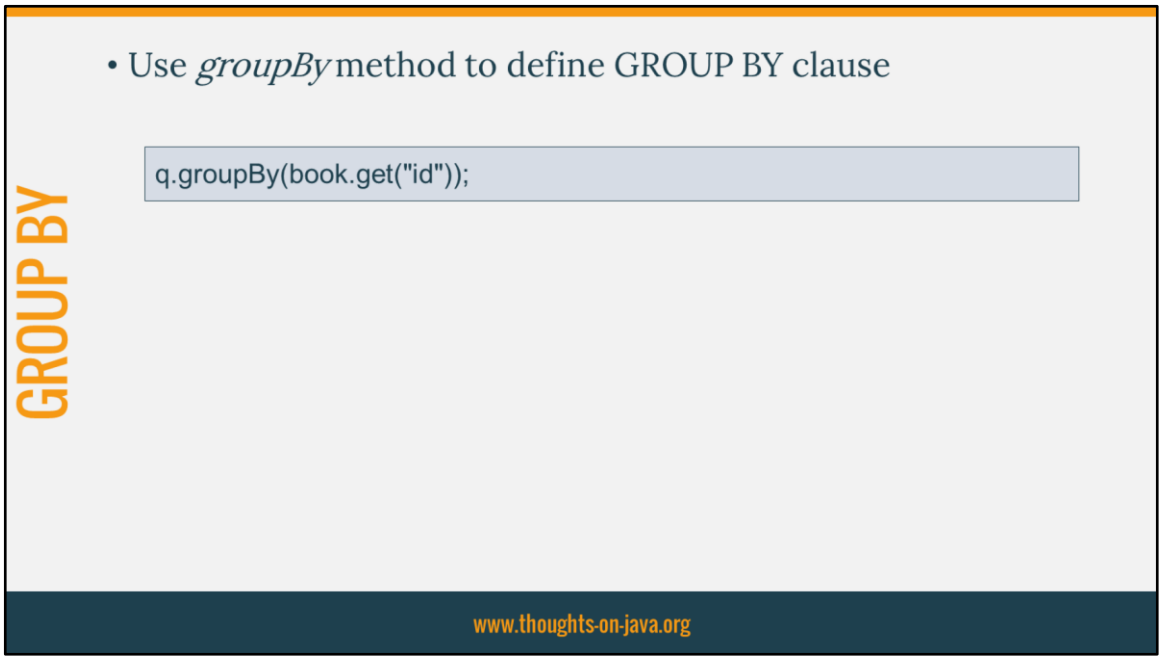


You probably now the constructor expression from JPQL queries. It describes a constructor that Hibernate will execute for each record returned from the database. You can do the same with the Criteria API. The *construct* method of the *CriteriaBuilder* interface defines the constructor call and expects the class Hibernate shall instantiate and a list of parameters. Similar to the JPQL constructor expression, Hibernate will provide the selected attributes in their defined order as parameters to the constructor. You, therefore, need to make sure that the class you want to instantiate has a matching constructor.

Demo

www.thoughts-on-java.org

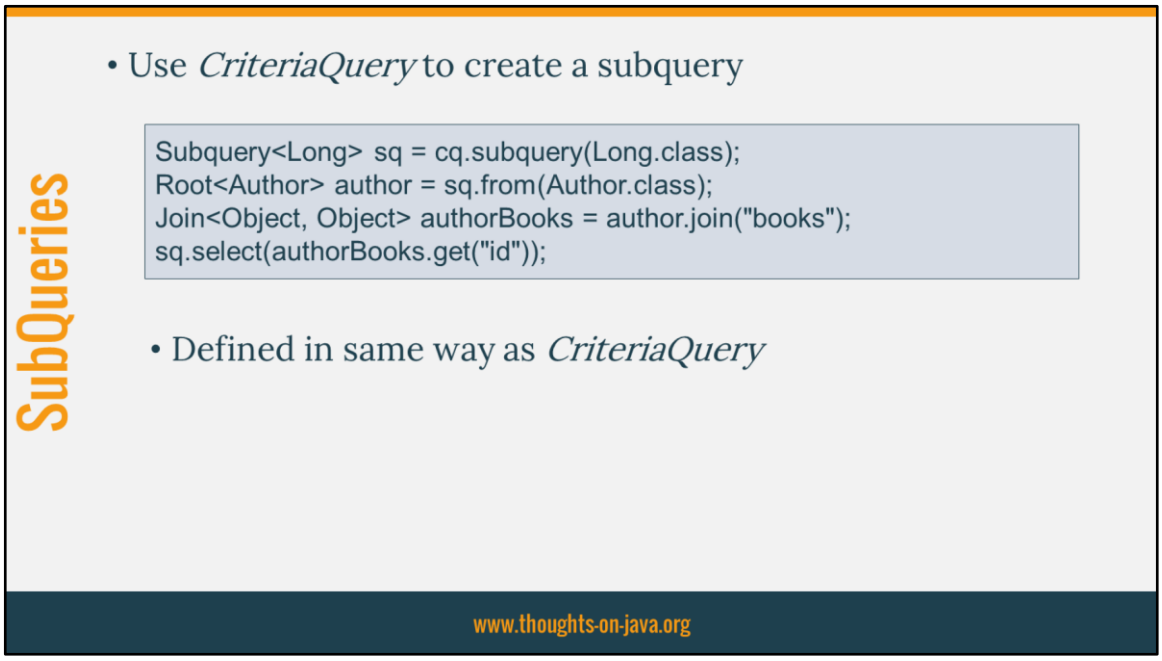
The next thing I want to show you are functions and the definition of a GROUP BY clause. Both of them are often used together.
The *CriteriaBuilder* interface provides a method for each function supported by the Criteria API. You can call them with references to the entity attributes on which they shall be applied and use them in the different clauses of your query. In the code snippet on this slide, I use the *count* function to count the number of authors who've written a specific book.



As you know from JPQL and SQL, the *count* function requires additional grouping of the attributes that are not provided to it. The same applies, of course, if you create the query with the Criteria API.
You can define the group clause by calling the *groupBy* method on the *CriteriaQuery*. In this code snippet, I reference the *id* attribute of the *Book* entity in the GROUP BY clause.

Demo

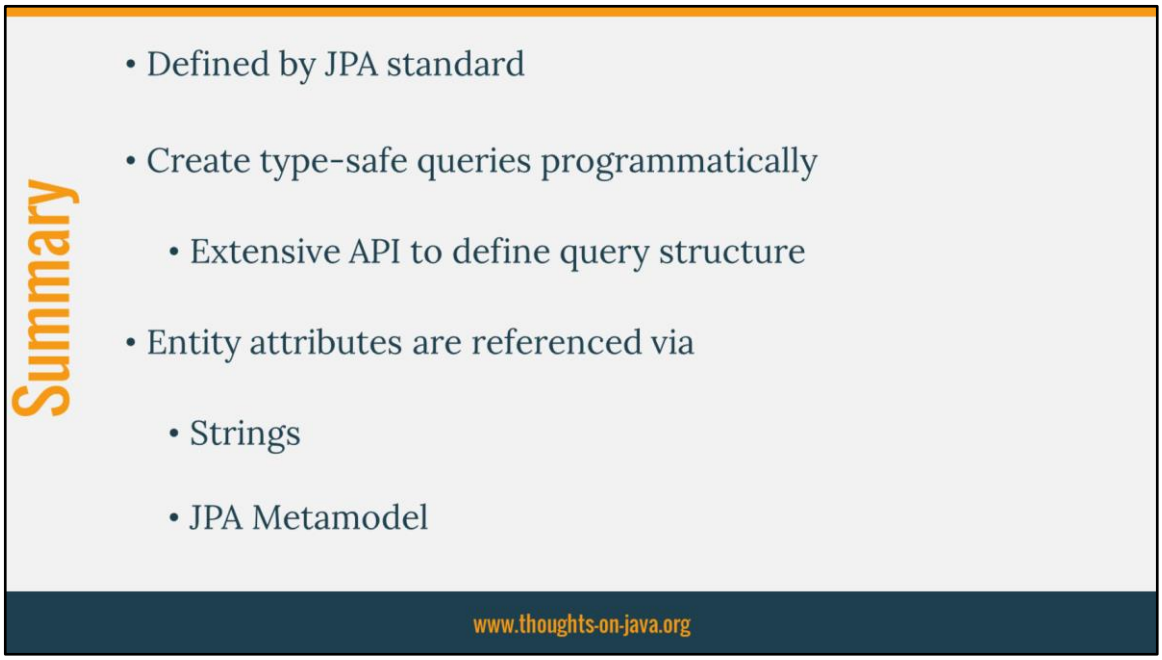
www.thoughts-on-java.org



The last thing I want to show you is the definition of a subquery. It's similar to the definition of a normal query. The only difference is that you have to call the *subquery* method on the *CriteriaQuery* interface to create a *Subquery* instance. You can then define it in the same way as a *CriteriaQuery*.

Demo

www.thoughts-on-java.org



The Criteria API that I showed you in this video is defined by the JPA standard. Hibernate's old Criteria API is deprecated and shouldn't be used.
JPA's Criteria API provides an extensive API to create type-safe queries programmatically. The API consists of multiple classes and interface that you can use to define the structure of your query. The entity attributes are referenced by *Strings* or attributes of the JPA Metamodel. I explain the Metamodel in the next video and recommend to use it. It provides a comfortable and type-safe way to reference all entity attributes.

Exercises

www.thoughts-on-java.org