

Managing Persistence Object



**IS 701 PENGEMBANGAN APLIKASI
ENTERPRISE**

**(C) 2010 NIKO IBRAHIM
FAKULTAS TEKNOLOGI INFORMASI
UNIVERSITAS KRISTEN MARANATHA**

Materi



- Entity Manager
- JPQL
- Query

Review JPA



- Selama ini, kita telah melihat bagaimana kemampuan JPA dalam memetakan object menjadi database relasional
- Berikutnya, kita akan melihat bagaimana cara JPA melakukan query terhadap object yang dipetakan tersebut.
- Di dalam JPA, terdapat Entity Manager yang merupakan ‘otak’ dalam memanipulasi entitas.
- Di dalam Entity Manager, terdapat API untuk meng-create, find, remove, maupun mensinkronisasi object dengan database.
- Selain itu, Entity Manager juga memungkinkan kita untuk melakukan query (JPQL) dan mekanisme locking (concurrency).

How to Query an Entity

@Entity

```
public class Book {  
    @Id  
    private Long id;  
    private String title;  
    private Float price;  
    private String description;  
    private String isbn;  
    private Integer nbOfPage;  
    private Boolean illustrations;  
    // Constructors, getters, setters  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        // 1-Create an instance of the Book entity  
        Book book = new Book();  
        book.setId(1234L);  
        book.setTitle("The Hitchhiker's Guide to the Galaxy");  
        book.setPrice(12.5F);  
        book.setDescription("Science fiction created by Douglas Adams.");  
        book.setIsbn("1-84023-742-2");  
        book.setNbOfPage(354);  
        book.setIllustrations(false);  
        // 2-Get an entity manager and a transaction  
        EntityManagerFactory emf =  
            Persistence.createEntityManagerFactory("BookStorePU");  
        EntityManager em = emf.createEntityManager();  
        EntityTransaction tx = em.getTransaction();  
        // 3-Persist the book to the database  
        tx.begin();  
        em.persist(book);  
        tx.commit();  
        // 4-Retrieve the book by its identifier  
        book = em.find(Book.class, 1234L);  
        System.out.println(book);  
        em.close();  
        emf.close();  
    }  
}
```

Persistence.xml



- Di dalam setiap aplikasi database, kita perlu mengetahui bagaimana cara melakukan koneksi dan menentukan database mana yang akan digunakan.
- Di dalam source code Main class tadi, terdapat "EntityManagerFactory" yang memiliki parameter "persistence unit" yaitu "BookStorePU". Persistence Unit inilah yang memberitahu kepada entity manager tipe database yang akan digunakan dan parameter koneksi lainnya.
- Persistence unit ini ditulis dalam satu file: **persistence.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name=" BookStorePU" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>com.niko.app.Book</class>
  <properties>
    <property name="eclipselink.target-database" value="DERBY"/>
    <property name="eclipselink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
    <property name="eclipselink.jdbc.url" value="jdbc:derby://localhost:1527/BookStoreDB"/>
    <property name="eclipselink.jdbc.user" value="APP"/>
    <property name="eclipselink.jdbc.password" value="APP"/>
  </properties>
</persistence-unit name>
```

Entity Manager



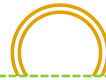
- The entity manager is a central piece in JPA. It manages the state and life cycle of entities as well as querying entities within a persistence context. The entity manager is responsible for creating and removing persistent entity instances and finding entities by their primary key. It can lock entities for protecting against concurrent access by using optimistic or pessimistic locking and can use JPQL queries to retrieve entities following certain criteria.
- When an entity manager obtains a reference to an entity, it is said to be *managed*. *Until* that point, the entity is seen as a regular POJO (i.e., detached).
- The strength of JPA is that entities can be used as regular objects by different layers of an application and become managed by the entity manager when you need to load or insert data into the database.
- When an entity is managed, you can carry out persistence operations, and the entity manager will automatically synchronize the state of the entity with the database. When the entity is detached (i.e., not managed), it returns to a simple POJO and can then be used by other layers (e.g., a JSF presentation layer) without synchronizing its state with the database.
- With the entity manager, the real work of persistence starts. EntityManager is an interface implemented by a persistence provider that will generate and execute SQL statements.

Obtaining an Entity Manager



- The entity manager is the central interface to interact with entities, but it first has to be obtained by an application.
- Depending on whether it is a **container-managed** environment or **application-managed** environment, the code can be quite different.
- For example, in a container-managed environment, the transactions are managed by the container. That means you don't need to explicitly write the commit or rollback, which you have to do in an application-managed environment.
- The term “application-managed” means an application is responsible for explicitly obtaining an instance of EntityManager and managing its life cycle (it closes the entity manager when finished, for example).

A Stateless EJB Injected with a Reference of an Entity Manager



```
@Stateless
public class BookBean {
    @PersistenceContext(unitName = "BookStorePU")
    private EntityManager em;
    public void createBook() {
        // Create an instance of book
        Book book = new Book();
        book.setId(1234L);
        book.setTitle("The Hitchhiker's Guide to the Galaxy");
        book.setPrice(12.5F);
        book.setDescription("Science fiction created by Douglas Adams.");
        book.setIsbn("1-84023-742-2");
        book.setNbOfPage(354);
        book.setIllustrations(false);
        // Persist the book to the database
        em.persist(book);
        // Retrieve the book by its identifier
        book = em.find(Book.class, 1234L);
        System.out.println(book);
    }
}
```

Persistence Context



- Before exploring the EntityManager API in detail, you need to understand a crucial concept: the *persistence context*.
- *A persistence context is a set of managed entity instances at a given time*: only one entity instance with the same persistent identity can exist in a persistence context.
- For example, if a Book instance with an ID of 1234 exists in the persistence context, no other book with this ID can exist within that same persistence context.
- Only entities that are contained in the persistence context are managed by the entity manager, meaning that changes will be reflected in the database.
- The entity manager updates or consults the persistence context whenever a method of the `javax.persistence.EntityManager` interface is called.
- For example, when a `persist()` method is called, the entity passed as an argument will be added to the persistence context if it doesn't already exist. Similarly, when an entity is found by its primary key, the entity manager first checks whether the requested entity is already present in the persistence context.
- The persistence context can be seen as a first-level cache. It's a short, live space where the entity manager stores entities before flushing the content to the database. Objects just live in the persistent context for the duration of the transaction.

Persistence Unit with a Set of Manageable Entities



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
<persistence-unit name="BookStorePU" transaction-type="RESOURCE_LOCAL">
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
<class>com.niko.app.Book</class>
<class>com.niko.app.Customer</class>
<class>com.niko.app.Address</class>
<properties>
<property name="eclipselink.target-database" value="DERBY"/>
<property name="eclipselink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
<property name="eclipselink.jdbc.url" value="jdbc:derby://localhost:1527/BookStoreDB"/>
<property name="eclipselink.jdbc.user" value="APP"/>
<property name="eclipselink.jdbc.password" value="APP"/>
</properties>
</persistence-unit>
</persistence>
```

Manipulating Entities

Method	Description
void persist (Object entity)	Makes an instance managed and persistent
<T> T find (Class<T> entityClass, Object primaryKey)	Searches for an entity of the specified class and primary key
<T> T getReference (Class<T> entityClass, Object primaryKey)	Gets an instance, whose state may be lazily fetched
void remove (Object entity)	Removes the entity instance from the persistence context and from the underlying database
<T> T merge (T entity)	Merges the state of the given entity into the current persistence context
void refresh (Object entity)	Refreshes the state of the instance from the database, overwriting changes made to the entity, if any
void flush ()	Synchronizes the persistence context to the underlying database
void clear ()	Clears the persistence context, causing all managed entities to become detached
void clear (Object entity)	Removes the given entity from the persistence context
boolean contains (Object entity)	Checks whether the instance is a managed entity instance belonging to the current persistence context

Contoh:

The Customer Entity with a One-Way, One-to-One Address

```
@Entity
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    @OneToOne (fetch = FetchType.LAZY)
    @JoinColumn(name = "address_fk")
    private Address address;
    // Constructors, getters, setters
}
```

```
@Entity
public class Address {
    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String city;
    private String zipcode;
    private String country;
    // Constructors, getters, setters
}
```

Persisting an Entity

Persisting a Customer with an Address



- Persisting an entity means inserting data into the database when the data doesn't already exist (otherwise an exception is thrown).
- To do so, it's necessary to create a new entity instance using the new operator, set the values of the attributes, bind one entity to another when there are associations, and finally call the **EntityManager.persist()** method

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
assertNotNull(customer.getId());
assertNotNull(address.getId());
```

Finding by ID

Finding a Customer by ID



- To find an entity by its identifier, you can use two different methods. The first is the **EntityManager.find()** method, which takes two parameters: **the entity class** and the **unique identifier**.
- If the entity is found, it is returned; if it is not found, a null value is returned.

```
Customer customer = em.find(Customer.class, 1234L)
if (customer != null) {
    // Process the object
}
```

Finding a Customer by Reference



- The second method is `getReference()`. It is very similar to the `find` operation, as it takes the same parameters, but it allows retrieval of a reference to an entity via its primary key, not its data.
- It is intended for situations where a managed entity instance is needed, but no data, other than potentially the entity's primary key being accessed.
- With `getReference()`, the state data is fetched lazily, which means that if you don't access state before the entity is detached, the data might not be there. If the entity is not found, an `EntityNotFoundException` is thrown.

```
try {  
    Customer customer = em.getReference(Customer.class, 1234L)  
    // Process the object  
} catch(EntityNotFoundException ex) {  
    // Entity not found  
}
```

Removing an Entity

Creating and Removing Customer and Address Entities



- An entity can be removed with the **EntityManager.remove()** method. Once removed, the entity is deleted from the database, is detached from the entity manager, and cannot be synchronized with the database anymore.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE", "UK");
customer.setAddress(address);
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
tx.begin();
em.remove(customer);
tx.commit();
// The data is removed from the database
// but the object is still accessible
assertNotNull(customer);
```

Synchronizing with the Database



- Until now the synchronization with the database has been done at commit time.
- The entity manager is a first-level cache, waiting for the transaction to be committed to flush the data to the database, but what happens when a customer and an address need to be inserted?

```
tx.begin();  
em.persist(customer);  
em.persist(address);  
tx.commit();
```

- All pending changes require an SQL statement, with two insert statements produced and made permanent only when the database **transaction commits**.
- For most applications, this automatic data synchronization is sufficient. Although at exactly which point in time the provider actually flushes the changes to the database is not known, you can be sure it happens inside the commit of the transaction.
- The database is synchronized with the entities in the persistence context, but data to the database can be **explicitly flushed** (flush), or entities **refreshed** with data from the database (refresh).
- If the data is flushed to the database at one point, and if later in the code the application calls the `rollback()` method, the flushed data will be taken out of the database.

Flushing Data



- With the **EntityManager.flush()** method, the persistence provider can be explicitly forced to flush the data, allowing a developer to manually trigger the same process used by the entity manager internally to flush the persistence context.

```
tx.begin();  
em.persist(customer);  
em.flush();  
em.persist(address);  
tx.commit();
```

Refreshing an Entity



- The `refresh()` method is used for data synchronization in the opposite direction of the `flush()`, meaning it overwrites the current state of a managed entity with data as it is present in the database.
- A typical case is where the `EntityManager.refresh()` method is used to undo changes that have been done to the entity in memory only.

```
Customer customer = em.find(Customer.class, 1234L)
assertEquals(customer.getFirstName(), "Antony");
customer.setFirstName("William");
em.refresh(customer);
assertEquals(customer.getFirstName(), "Antony");
```

Content of the Persistence Context Contains



- Entities are either managed or not by the entity manager. The `EntityManager.contains()` method returns a Boolean and allows checking of whether a particular entity instance is currently managed by the entity manager within the current persistence context.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
assertTrue(em.contains(customer));
tx.begin();
em.remove(customer);
tx.commit();
assertFalse(em.contains(customer));
```

Content of the Persistence Context

Clear and Detach



- The `clear()` method is straightforward: it empties the persistence context, causing all managed entities to become detached.
- The `detach(Object entity)` method removes the given entity from the persistence context.
- Changes made to the entity will not be synchronized to the database after such eviction has taken place

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
assertTrue(em.contains(customer));
em.detach(customer);
assertFalse(em.contains(customer));
```

Merging an Entity



- A detached entity is no longer associated with a persistence context. If you want to manage it, you need to merge it

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");
tx.begin();
em.persist(customer);
tx.commit();
em.clear();
// Sets a new value to a detached entity
customer.setFirstName("William");
tx.begin();
em.merge(customer);
tx.commit();
```

Updating an Entity



- Updating an entity is simple, yet at the same time it can be confusing to understand. As you've just seen, you can use the **EntityManager.merge()** to attach an entity and synchronize its state with the database.
- But if an entity is currently managed, changes to it will be reflected in the database automatically. If not, you will need to explicitly call **merge()**.

```
Customer customer = new Customer("Antony", "Balla", "tballa@mail.com");  
tx.begin();  
em.persist(customer);  
customer.setFirstName("Williman");  
tx.commit();
```



JPQL & Query

- **JPQL**
 - Select
 - From
 - Where
 - Order By
 - Group By & Having
 - Bulk Delete
 - Bulk Update
- **Query**
 - Dynamic Query
 - Named Query
 - Native Query

JPQL



- JPQL is used to define searches against persistent entities independent of the underlying database. JPQL is a query language that takes its roots in the syntax of Standard Query Language (SQL), which is the standard language for database interrogation.
- But the main difference is that in SQL the results obtained are in the form of rows and columns (tables), whereas JPQL uses an entity or a collection of entities.
- JPQL syntax is object oriented and therefore more easily understood by developers whose experience is limited to object-oriented languages. Developers manage their entity domain model, not a table structure, by using the dot notation (e.g., `myClass.myAttribute`).
- Under the hood, JPQL uses the mechanism of mapping to transform a JPQL query into language comprehensible by a SQL database. The query is executed on the underlying database with SQL and JDBC calls, and then entity instances have their attributes set and are returned to the application—all in a very simple and powerful manner using a rich query syntax.

Examples



```
SELECT b
FROM Book b
```

- Instead of selecting from a table, JPQL selects entities, here Book.
- The FROM clause is also used to give an alias to the entity: b is an alias for Book. The SELECT clause of the query indicates that the result type of the query is the b entity (the Book). Executing this statement will result in a list of zero or more Book instances

```
SELECT b
FROM Book b
WHERE b.title = "H2G2"
```

- The alias is used to navigate across entity attributes through the dot operator. Since the Book entity has a persistent attribute named title of type String, b.title refers to the title attribute of the Book entity.

Select



- Sintaks:

```
SELECT <select expression>  
FROM <from clause>  
[WHERE <conditional expression>]  
[ORDER BY <order by clause>]  
[GROUP BY <group by clause>]  
[HAVING <having clause>]
```

- Contoh:

```
SELECT c  
FROM Customer c
```

```
SELECT c.firstName  
FROM Customer c
```

```
SELECT c.firstName, c.lastName  
FROM Customer c
```

```
SELECT c.address  
FROM Customer c
```

```
SELECT DISTINCT c  
FROM Customer c
```

```
SELECT DISTINCT c.firstName  
FROM Customer c
```

Aggregate Function



- The result of a query may be the result of an aggregate function applied to a path expression.
- The following aggregate functions can be used in the SELECT clause: AVG, COUNT, MAX, MIN, SUM. The results may be grouped in the GROUP BY clause and filtered using the HAVING clause.

```
SELECT COUNT(c)  
FROM Customer c
```

Where



- The WHERE clause of a query consists of a conditional expression used to restrict the result of a SELECT, UPDATE, or DELETE statement.
- The WHERE clause can be a simple expression or set of conditional expressions to filter the query in a sophisticated way.
- Contoh:

```
SELECT c  
FROM Customer c  
WHERE c.firstName = 'Vincent'
```

```
SELECT c  
FROM Customer c  
WHERE c.age > 18
```

```
SELECT c  
FROM Customer c  
WHERE c.firstName = 'Vincent' AND c.address.country = 'France'
```

```
SELECT c  
FROM Customer c  
WHERE c.age NOT BETWEEN 40 AND 50
```

```
SELECT c  
FROM Customer c  
WHERE c.email LIKE '%mail.com'
```

```
SELECT c  
FROM Customer c  
WHERE c.address.country IN ('USA', 'Portugal')
```

Binding Parameters



- Until now, the WHERE clauses shown herein only used fixed values. In an application, queries frequently depend on parameters.
- JPQL supports two types of parameter-binding syntax, allowing dynamic changes to the restriction clause of a query: **positional** and **named parameters**.
- ***Positional parameters** are designated by the question mark (?) followed by an integer (e.g., ?1).* When the query is executed, the parameter numbers that should be replaced need to be specified.

```
SELECT c
FROM Customer c
WHERE c.firstName = ?1 AND c.address.country = ?2
```

- ***Named parameters** can also be used and are designated by a String identifier that is prefixed by the colon (:) symbol.* When the query is executed, the parameter names that should be replaced need to be specified.

```
SELECT c
FROM Customer c
WHERE c.firstName = :fname AND c.address.country = :country
```

Subqueries



- A subquery is a SELECT query that is embedded within a conditional expression of a WHERE or HAVING clause.
- The results of the subquery are evaluated and interpreted in the conditional expression of the main query.
- To retrieve the youngest customers from the database, a subquery with a MIN(age) is first executed and its result evaluated in the main query.

```
SELECT c  
FROM Customer c  
WHERE c.age = (SELECT MIN(c. age) FROM Customer c)
```

Order By



- The ORDER BY clause allows the entities or values that are returned by a SELECT query to be ordered.
- The ordering applies to the entity attribute specified in this clause followed by the ASC or DESC keyword.
- The keyword ASC specifies that ascending ordering be used; DESC, the inverse, specifies that descending ordering be used. Ascending is the default and can be omitted.

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC
```

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC, c.address.country ASC
```

Group By and Having



- The GROUP BY construct enables the aggregation of result values according to a set of properties.
- The entities are divided into groups based on the values of the entity field specified in the GROUP BY clause.
- To group customers by country and count them, use the following query:

```
SELECT c.address.country, count(c)
FROM Customer c
GROUP BY c.address.country
```

- The GROUP BY defines the grouping expressions (c.address.country) over which the results will be aggregated and counted (count(c)).

Having



- The HAVING clause defines an applicable filter after the query results have been grouped, similar to a secondary WHERE clause filtering the result of the GROUP BY.
- Using the previous query, by adding a HAVING clause, a result of only those countries where the number of customers is greater than 1,000 can be achieved.

```
SELECT c.address.country, count(c)
FROM Customer c
GROUP BY c.address.country
HAVING count(c) > 100
```

Bulk Delete



- JPQL performs bulk delete operations across multiple instances of a specific entity class.
- These are used to delete a large number of entities in a single operation. The DELETE statement looks like the SELECT statement, as it can have a restricting WHERE clause and take parameters.
- As a result, the number of entity instances affected by the operation is returned.
- As an example, to delete all customers younger than 18, you can use a bulk removal via a DELETE statement:

```
DELETE FROM Customer c  
WHERE c.age < 18
```

Bulk Update



- Bulk updates of entities are accomplished with the UPDATE statement, setting one or more attributes of the entity subject to conditions in the WHERE clause.
- Rather than deleting all the young customers, their first name can be changed to “too young” with the following statement:

```
UPDATE Customer c  
SET c.firstName = 'TOO YOUNG'  
WHERE c.age < 18
```

Queries



- You've seen the JPQL syntax and how to describe statements using different clauses (SELECT, FROM, WHERE, etc.).
- But how do you integrate a JPQL statement to your application?
- The answer: through queries.
- JPA 2.0 has four different types of queries that can be used in code, each with a different purpose:
 - *Dynamic queries*: This is the simplest form of queries, consisting of nothing more than a JPQL query string dynamically specified at runtime.
 - *Named queries*: Named queries are static and unchangeable.
 - *Native queries*: This type of query is useful to execute a native SQL statement instead of a JPQL statement.
 - *Criteria API*: JPA 2.0 introduces this new concept.

EntityManager Methods for Creating Queries



Method	Description
Query createQuery (String jpqlString)	Creates an instance of Query for executing a JPQL statement for dynamic queries
Query createQuery (QueryDefinition qdef)	Creates an instance of Query for executing a criteria query
Query createNamedQuery (String name)	Creates an instance of Query for executing a named query (in JPQL or in native SQL)
Query createNativeQuery (String sqlString)	Creates an instance of Query for executing a native SQL statement
Query createNativeQuery (String sqlString, Class resultClass)	Creates an instance of Query for executing a native SQL statement passing the class of the expected results

Dynamic Queries



- Dynamic queries are defined on the fly as needed by the application. To create a dynamic query, use the **EntityManager.createQuery()** method, which takes a String as a parameter that represents a JPQL query.
- In the following code, the JPQL query selects all the customers from the database. As the result of this query is a list, the **getResultList()** method is used and returns a list of Customer entities (List<Customer>). However, if you know that your query only returns a single entity, use the **getSingleResult()** method. It returns a single entity and avoids the work of pulling it off a list.

```
Query query = em.createQuery("SELECT c FROM Customer c");  
List<Customer> customers = query.getResultList();
```

Dynamic Queries (continue)



- This query string can also be dynamically created by the application, which can then specify a complex query at runtime not known ahead of time.
- String concatenation is used to construct the query dynamically depending on the criteria.

```
String jpqlQuery = "SELECT c FROM Customer c";  
if (someCriteria)  
    jpqlQuery += " WHERE c.firstName = 'Vincent'";  
query = em.createQuery(jpqlQuery);  
List<Customer> customers = query.getResultList();
```

Dynamic Queries (continue)



- The previous query retrieves customers named Vincent, but you might want to introduce a parameter for the first name. There are two possible choices for passing a parameter: **using names** or **positions**. In the following example, a **named parameter called :fname** (note the : symbol) is used in the query and bound with the setParameter method:

```
jpqlQuery = "SELECT c FROM Customer c";  
if (someCriteria)  
jpqlQuery += " where c.firstName = :fname";  
query = em.createQuery(jpqlQuery);  
query.setParameter("fname", "Vincent");  
List<Customer> customers = query.getResultList();
```

- The code using **a position parameter** would look like the following:

```
jpqlQuery = "SELECT c FROM Customer c";  
if (someCriteria)  
jpqlQuery += " where c.firstName = ?1";  
query = em.createQuery(jpqlQuery);  
query.setParameter(1, "Vincent");  
List<Customer> customers = query.getResultList();
```

Dynamic Queries (continue)



- If you need to use pagination to display the list of customers by chunks of ten, you can use the **setMaxResults** method as follows:

```
Query query = em.createQuery("SELECT c FROM Customer c");  
query.setMaxResults(10);  
List<Customer> customers = query.getResultList();
```

Named Queries



- **Named queries** are different from dynamic queries in that they are **static and unchangeable**.
- In addition to their static nature, which does not allow the flexibility of a dynamic query, named queries can be **more efficient** to execute because the persistence provider can translate the JPQL string to SQL once the application starts, rather than every time the query is executed.
- Named queries are static queries expressed in metadata inside either a `@NamedQuery` annotation or the XML equivalent.
- To define these reusable queries, annotate an entity with the `@NamedQuery` annotation, which takes two elements: **the name of the query** and **its content**.

The Customer Entity Defining Named Queries



```
@Entity
@NamedQueries({
    @NamedQuery(name = "findAll", query="select c from Customer c"),
    @NamedQuery(name = "findVincent", query="select c from Customer c where c.firstName = 'Vincent'"),
    @NamedQuery(name = "findWithParam", query="select c from Customer c where c.firstName = :fname")
})
public class Customer {
    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private Integer age;
    private String email;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
    // Constructors, getters, setters
}
```

Named Queries (continue)



- The way to execute these named queries resembles the way dynamic queries are used.
- The `EntityManager.createNamedQuery()` method is invoked and passed to the query name defined by the annotations.
- This method returns a Query that can be used to set parameters, the max results, fetch modes, and so on.
- To execute named queries:

```
Query query = em.createNamedQuery("findAll");  
List<Customer> customers = query.getResultList();
```

```
Query query = em.createNamedQuery("findWithParam");  
query.setParameter("fname", "Vincent");  
query.setMaxResults(3);  
List<Customer> customers = query.getResultList();
```

The Customer Entity Defining a Named Query with a Constant



```
@Entity
@NamedQuery(name = Customer.FIND_ALL, query="select c from Customer c"),
public class Customer {
    public static final String FIND_ALL = "Customer.findAll";
    // Attributes, constructors, getters, setters
}
```

- The **FIND_ALL** constant identifies the findAll query in a nonambiguous way by prefixing the name of the query with the name of the entity.
- The same constant is then used in the @NamedQuery annotation, and you can use this constant to execute the query as follows:

```
Query query = em.createNamedQuery(Customer.FIND_ALL);
List<Customer> customers = query.getResultList();
```

Native Queries



- JPQL has a very rich syntax that allows you to handle entities in any form and in a portable way across databases.
- JPA enables you to use specific features of a database by using native queries.
- Native queries take a native SQL statement (SELECT, UPDATE, or DELETE) as the parameter and return a Query instance for executing that SQL statement.
- However, native queries are not expected to be portable across databases.

```
Query query = em.createNativeQuery(  
    "SELECT * FROM t_customer", Customer.class);  
List<Customer> customers = query.getResultList();
```

The Customer Entity Defining a Native Named Query



- Named, native queries are defined using the **@NamedNativeQuery** annotation, which must be placed on any entity.
- Like JPQL named queries, the name of the query must be unique within the persistence unit.

```
@Entity
@NamedNativeQuery(name = "findAll", query="select * from t_customer")
@Table(name = "t_customer")
public class Customer {
    // Attributes, constructors, getters, setters
}
```