JAVA FULL STACK DEVELOPMENT PROGRAM

Session 12: Hibernate 2

OUTLINE

- SessionFactory & Session
- Entity Management
- Relationship mapping
- Query with Hibernate

HIBERNATE SESSION & SESSION FACTORY

- The session object provides an interface between the application and data stored in the database.
    - A Session is a light weight and a non-threadsafe object that represents a single unit-of-work with the database.
    - It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria.
- SessionFactory is Hibernate's concept of a single datastore and is thread-safe so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database.

SESSION

- How hibernate session work?
    - Unit of Work (session-per-operation) — a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.
        - In other words, its a series of operations we wish to carry out against the database together
    - Do not open and close a session for every simple database call in a single thread.
        - Database calls in an application are made using a planned sequence
        - They are grouped into atomic units of work
        - This also means that auto-commit after every single SQL statement is useless in an application

CLOSE

- sessionFactory.getCurrentSession()
  - We'll obtain a "current session" which is bound to the lifecycle of the transaction and will be automatically flushed and closed when the transaction ends (commit or rollback).
- sessionFactory.openSession()
  - We'll have to manage the session yourself and to flush or close it “manually"

ENTITY MANAGEMENT

- Apart from object-relational mapping itself, one of the problems that Hibernate was intended to solve is the problem of managing entities during runtime.

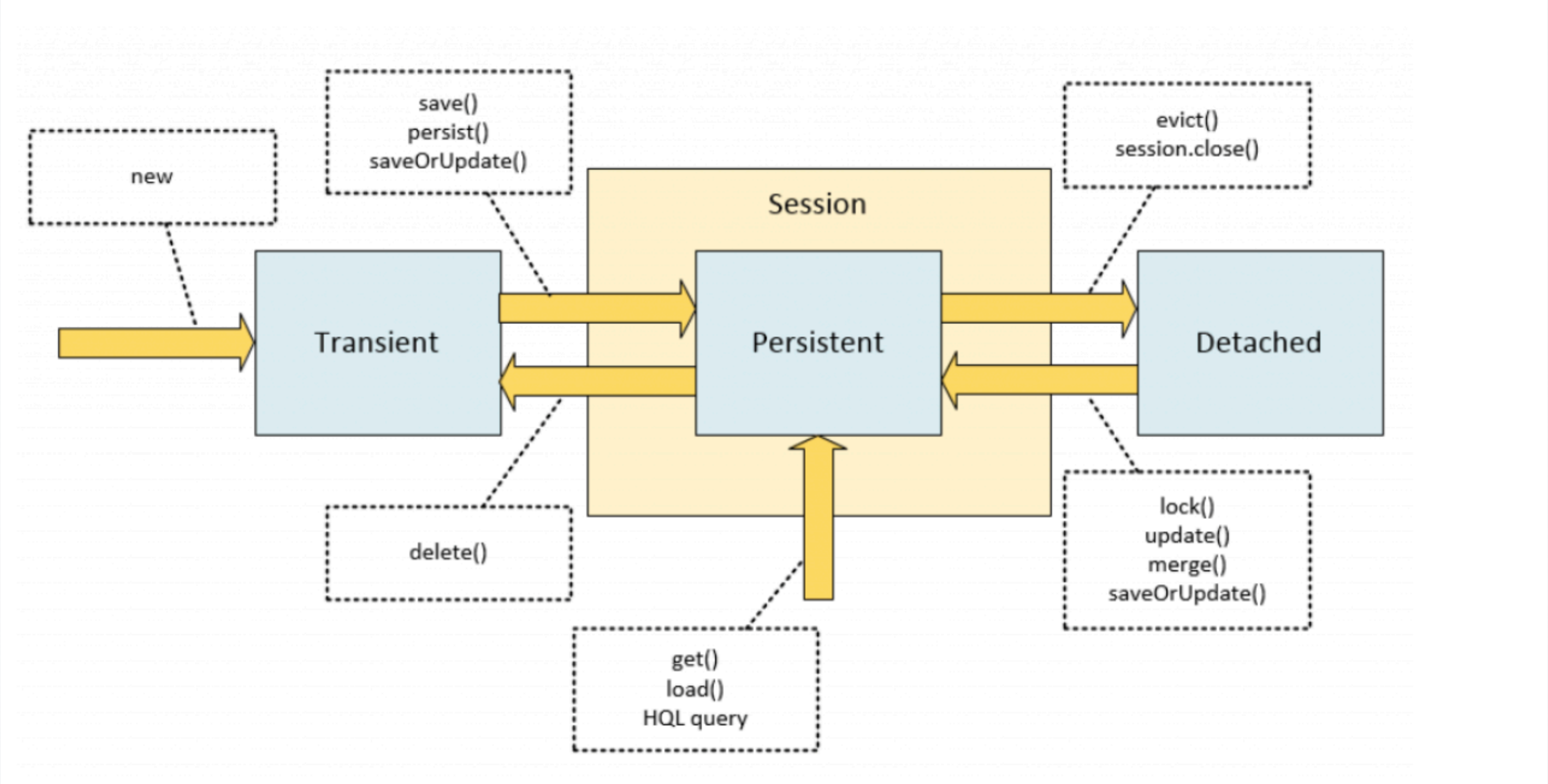
- The notion of "persistence context" is Hibernate's solution to this problem.

- Persistence context can be thought of as a container or a first-level cache for all the objects that you loaded or saved to a database during a session.

STATES OF ENTITY INSTANCES

- Any entity instance in your application appears in one of the three main states in relation to the *Session* persistence context:
  - *transient* — this instance is not, and never was, attached to a *Session*; this instance has no corresponding rows in the database; it's usually just a new object that you have created to save to the database;
  - *persistent* — this instance is associated with a unique *Session* object; upon flushing the *Session* to the database, this entity is guaranteed to have a corresponding consistent record in the database;
  - *detached* — this instance was once attached to a *Session* (in a *persistent* state), but now it's not; an instance enters this state if you evict it from the context, clear or close the Session, or put the instance through serialization/deserialization process.

STATES OF ENTITY INSTANCES



STATES OF ENTITY INSTANCES

- When the entity instance is in the *persistent* state, all changes that you make to the ***mapped*** fields of this instance will be applied to the corresponding database records and fields upon ***flushing*** the *Session*.

- The *persistent* instance can be thought of as "online", whereas the *detached* instance has gone "offline" and is not monitored for changes.

- This means that when you change fields of a *persistent* object, you *don't* have to call *save*, *update* or any of those methods to get these changes to the database: all you need is to ***commit*** the transaction, or ***flush*** or ***close*** the session, when you're done with it.

SESSION INTERFACE

- To Persistent State:
  - Save
  - Persist
  - Update
  - Merge
  - saveOrUpdate
- Note: These methods do not immediately result in the corresponding SQL *UPDATE* or *INSERT* statements. The actual saving of data to the database occurs on committing the transaction or flushing the *Session*.

PERSIST

- The *persist* method is intended for adding a new entity instance to the persistence context,
    - i.e. transitioning an instance from transient to *persistent* state.
    - We usually call it when we want to add a record to the database (persist an entity instance)
- The *persist* method has *void* return type. It operates on the passed object "in place", changing its state.
    - The object passed in now actually pointing to the persisted object
- Note: This method does NOT guarantee that the id of the object will be generated after calling the method. It follows JPA specification.

SAVE

- The method strictly states that it persists the instance, "first assigning a generated identifier".
  - The method is guaranteed to return the *Serializable* value of this identifier.
- The method has a return type of Serializable
- The reference of the passed in object pointing to the persisted object.
- Note: it does not conform to the JPA specification.

MERGE

- The main intention of the *merge* method is to update a *persistent* entity instance with new field values from a *detached* entity instance
    - Suppose we have a RESTful interface with a method for retrieving an JSON-serialized object by its id to the caller and a method that receives an updated version of this object from the caller.
- An entity that passed through such serialization/deserialization will appear in a *detached* state. So the *merge* method does exactly that:
    - Finds an entity instance by id taken from the passed object
    - Copies fields from the passed object to this instance
    - Returns newly updated instance
- The return type of the method is an Object — It is the object loaded into the persistent state and updated, not the object passed as the argument.
- Note: It follows JPA specification.

UPDATE

- It acts almost same as Save and Persist method, with small different:
  - It acts upon passed object (its return type is *void*)
  - The *update* method transitions the passed object from *detached* to *persistent* state
  - This method throws an exception if you pass it a *transient* entity
- Note: it does not confirm to the JPA specification.

SAVEORUPDATE

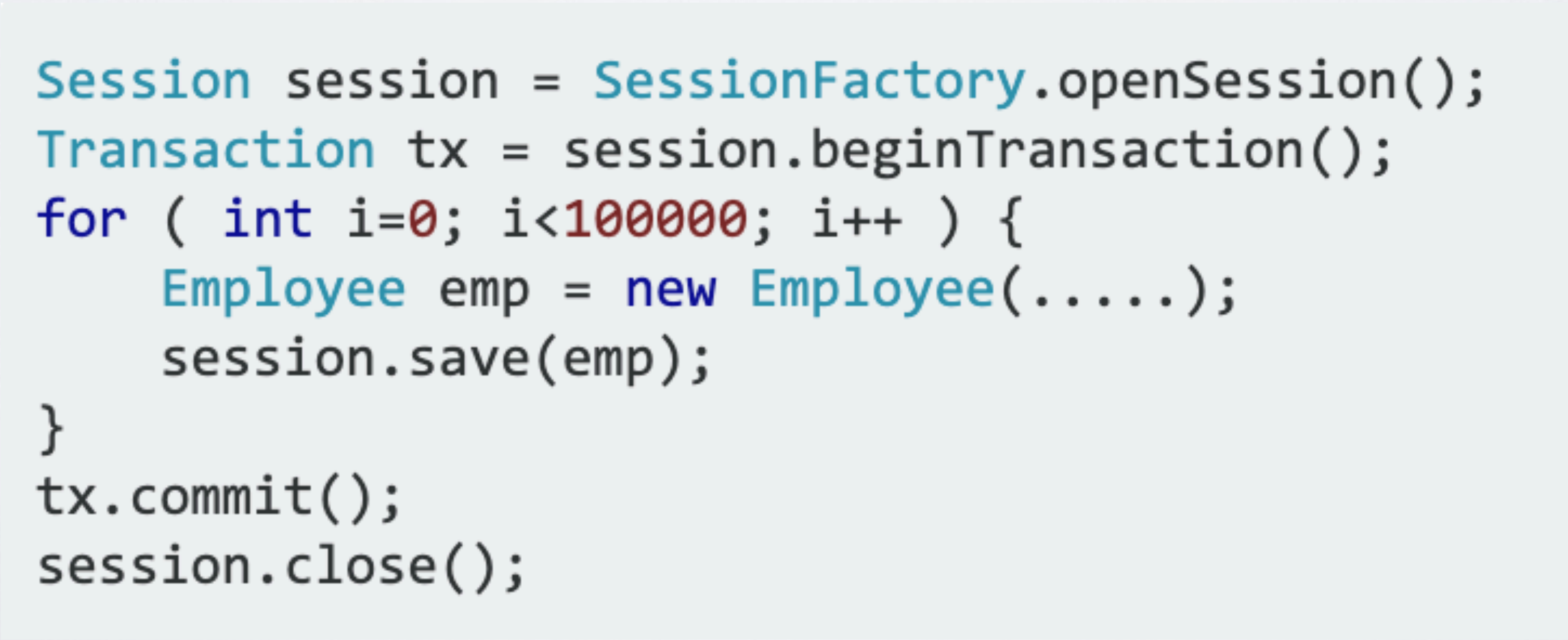
- Similar to *update*, it also may be used for reattaching instances
- The main difference of *saveOrUpdate* method is that it does not throw exception when applied to a *transient* instance; instead, it makes this *transient* instance *persistent*.
- Note: it does not conform to the JPA specification.

SESSION INTERFACE

- To detached state:
  - close
  - evict
  - Serialize and Deserialize

FLUSH

- It is used to synchronize session data with database.
  - When we call session.flush(), the statements are executed in database but it will not committed.
  - seesion.flush() just executes the statements in database (but not commits) and statements are NOT IN MEMORY anymore
- Why do we need to call flush()? Consider the following code:



```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Employee emp = new Employee(.....);
    session.save(emp);
}
tx.commit();
session.close();
```

EVICT

- evict() — remove the object from persistent state.
    - After detaching the object from the session, any change to object will not be persisted
- Why do we need evict?

RELATIONSHIP MAPPING

- In Java, how can we represent the relationship between two class?
  - Inheritance
  - Aggregation / Composition
- In Database, how can we represent the relationship between two table?
  - One to One
  - One to Many / Many to One
  - Many to Many

RELATIONSHIP MAPPING

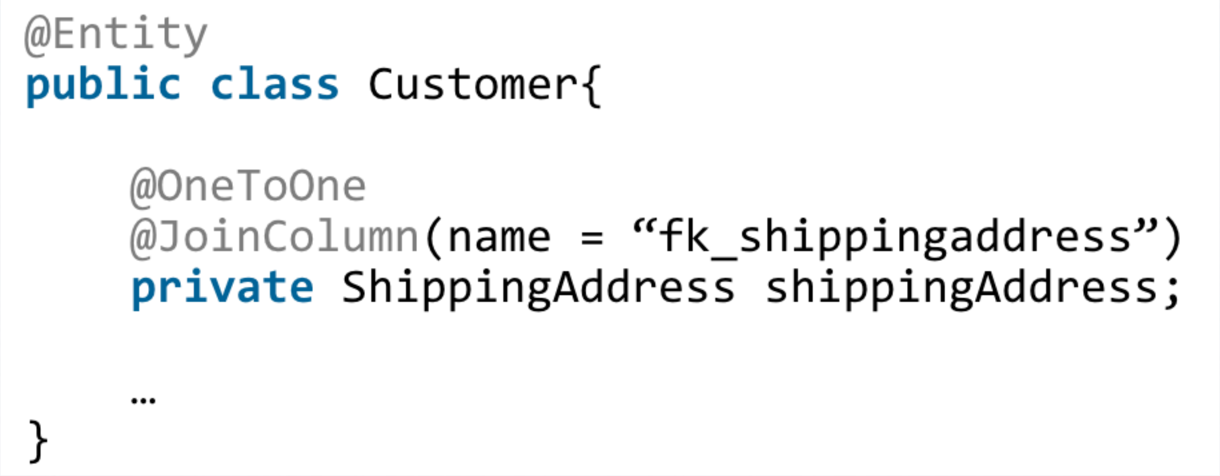
- Association mappings are one of the key features of JPA and Hibernate.
- They model the relationship between two database tables as attributes in your domain model. (Aggregation)
- Three types of mapping supported:
  - one-to-one
  - many-to-one / one-to-many
  - many-to-many
- Note: We can map each of them as a uni- or bidirectional association.
  - That has no impact on your database mapping, but it defines in which direction you can use the relationship in your domain model and HQL or Criteria queries

ONE TO ONE

- One-to-one relationships are rarely used in relational table models.
    - An example for a one-to-one association could be a *Customer* and the Current*Address*. Each *Customer* has exactly one *ShippingAddress* and each *ShippingAddress* belongs to one *Customer*.
    - On the database level, this mapped by a foreign key column either on the *ShippingAddress* or the *Customer* table
- We can have unidirectional or bidirectional mapping between those two entity.

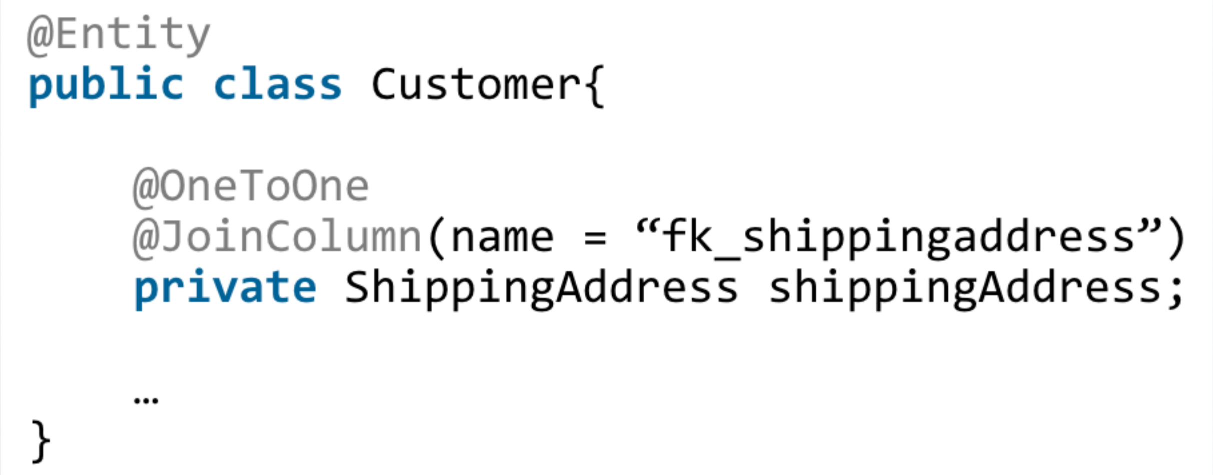
ONE TO ONE

- Unidirectional mapping

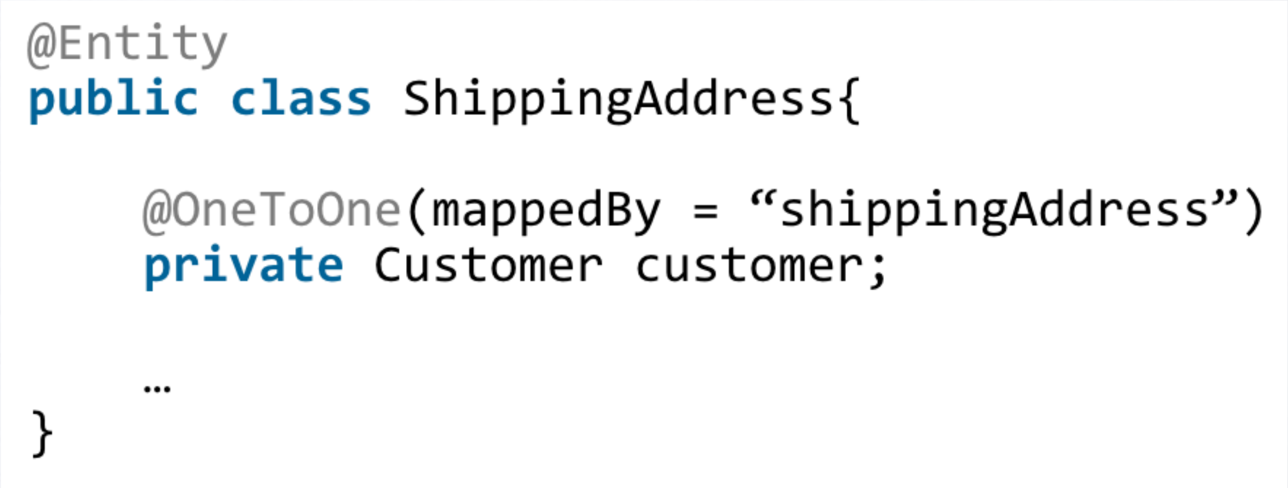


```
@Entity
public class Customer{

    @OneToOne
    @JoinColumn(name = “fk_shippingaddress”)
    private ShippingAddress shippingAddress;

    …
}
```

- Bidirectional mapping



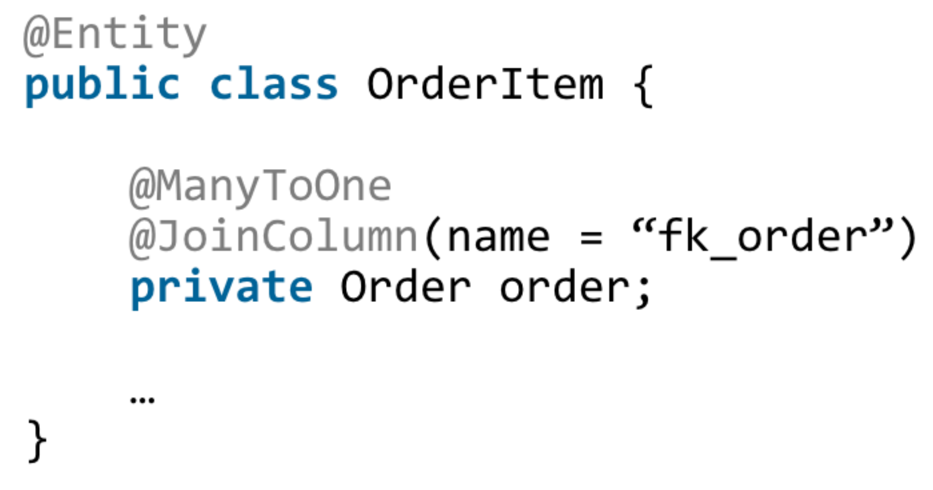
```
@Entity
public class Customer{

    @OneToOne
    @JoinColumn(name = “fk_shippingaddress”)
    private ShippingAddress shippingAddress;

    …
}
```



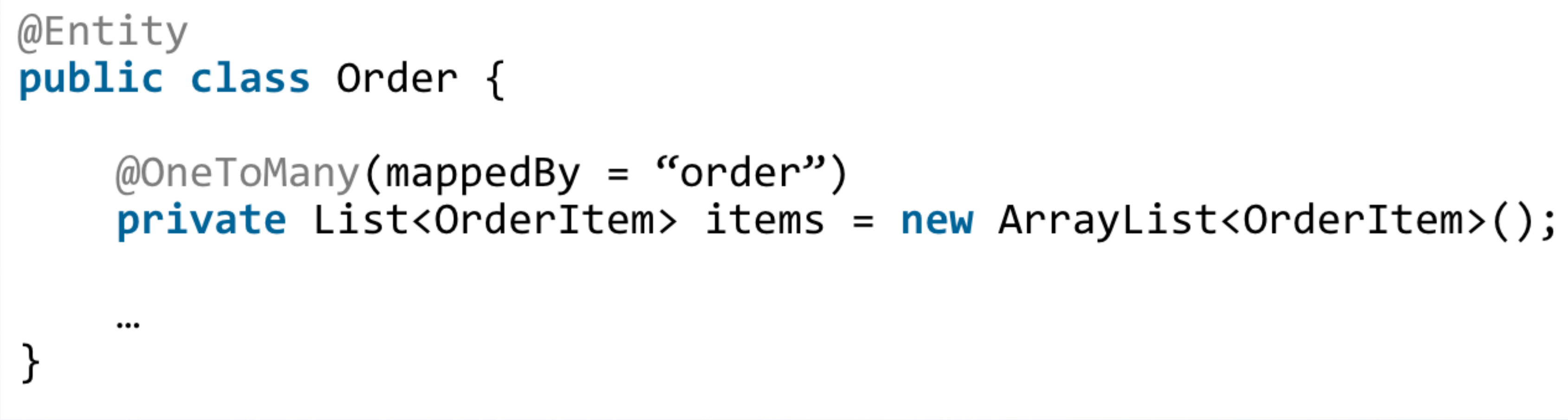
```
@Entity
public class ShippingAddress{

    @OneToOne(mappedBy = “shippingAddress”)
    private Customer customer;

    …
}
```

MANY TO ONE

- For example, An order consists of multiple items, but each item belongs to only one order.
- On database side, we need to store the primary key of the Order record as a foreign key in the OrderItem table
- With Hibernate or JPA, we can use @ManyToOne and @OneToMany



```
@Entity
public class OrderItem {

    @ManyToOne
    @JoinColumn(name = “fk_order”)
    private Order order;

    …
}
```



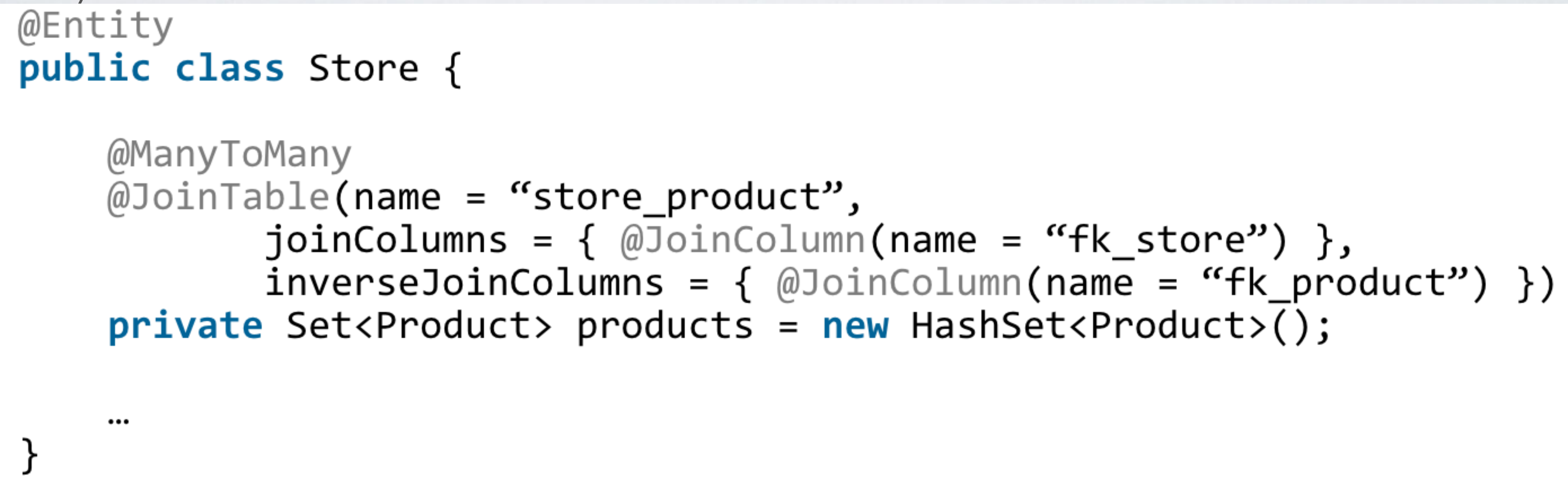
```
@Entity
public class Order {

    @OneToMany(mappedBy = “order”)
    private List<OrderItem> items = new ArrayList<OrderItem>();

    …
}
```

MANY TO MANY

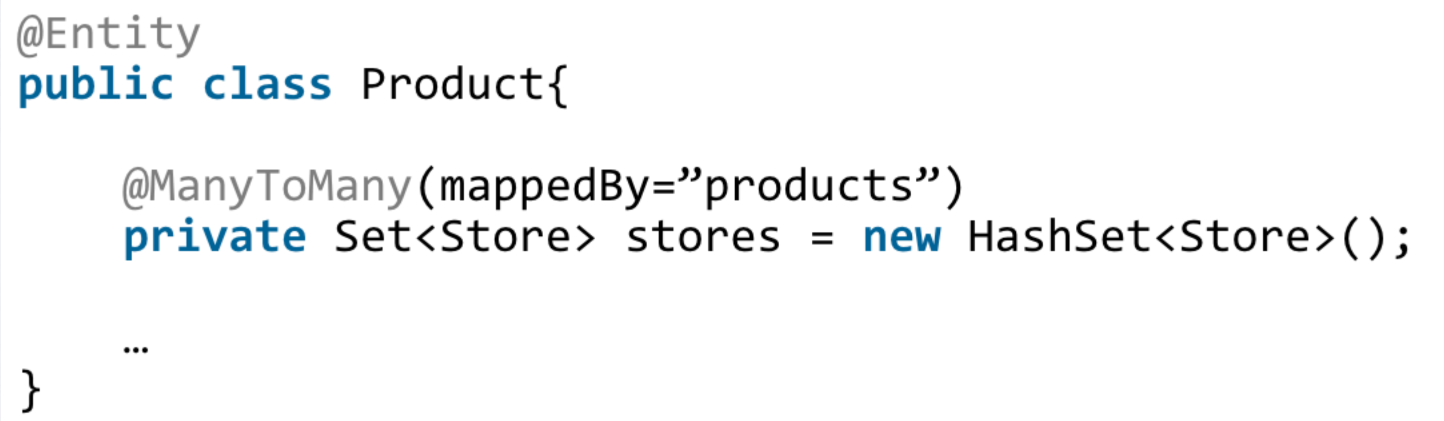
- A typical example for such a many-to-many relationship are *Products* and *Stores*.
    - Each *Store* sells multiple *Products* and each *Product* gets sold in multiple *Stores*.
- On database side, it requires an additional conjunction table which contains the primary key pairs of the associated entities.
- With Hibernate or JPA, we don't need to map this conjunction table to an entity.

MANY TO MANY

- @ManyToMany



```
@Entity
public class Store {

    @ManyToMany
    @JoinTable(name = “store_product”,
           joinColumns = { @JoinColumn(name = “fk_store”) },
           inverseJoinColumns = { @JoinColumn(name = “fk_product”) })
    private Set<Product> products = new HashSet<Product>();


    …
}
```



```
@Entity
public class Product{

    @ManyToMany(mappedBy=”products”)
    private Set<Store> stores = new HashSet<Store>();


    …
}
```

- If we don’t provide any additional information in @ManyToMany, Hibernate uses its default mapping which expects an association table with the name of both entities and the primary key attributes of both entities. In this case, Hibernate uses the *Store_Product* table with the columns *store_id* and *product_id*.

RELATIONSHIP MAPPING

- What do we miss?
  - With Many-to-Many or Many-to-One relationship, how should we add an item into the collection?
    - store.getProducts().add(product) — is it ok to use the statement in our business logic?

HIBERNATE QUERY

- There are several ways to query database using Hibernate:
    - HQL — Hibernate Query Language
    - Criteria
    - Native Query

HQL

- Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.

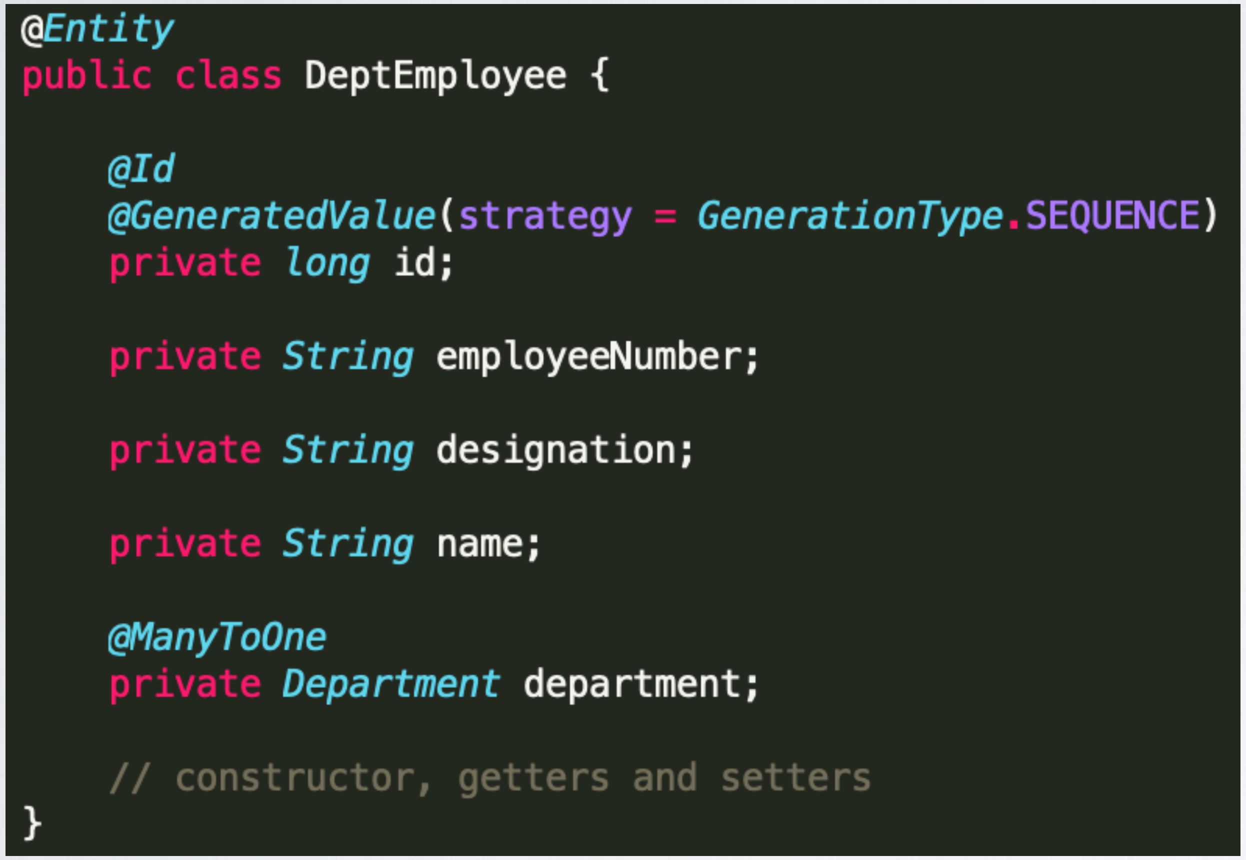
- HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.

- Although the SQL statements are also supported by Hibernate, we should use HQL as much as possible to achieve database portability

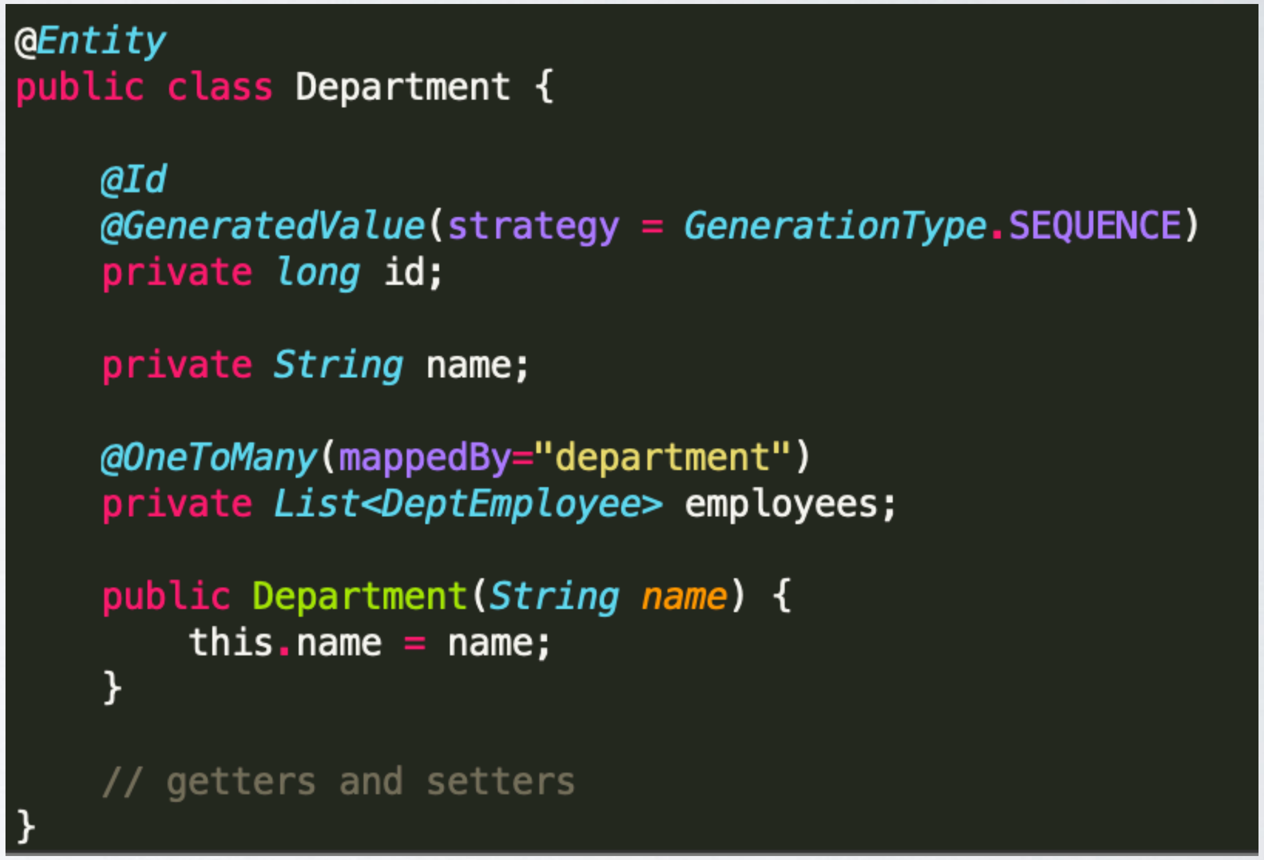
- Note: Keywords like SELECT, FROM, and WHERE, etc., are NOT case sensitive, but properties like table and column names are *case sensitive* in HQL

- Link to full reference: https://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/queryhql.html

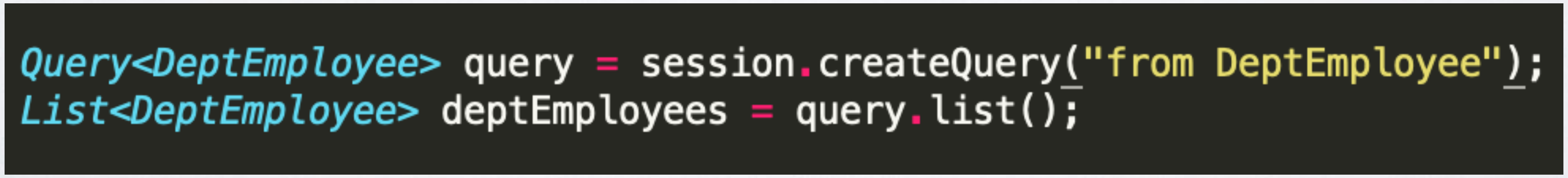
HQL



```java
@Entity
public class DeptEmployee {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String employeeNumber;

    private String designation;

    private String name;

    @ManyToOne
    private Department department;

    // constructor, getters and setters
}
```

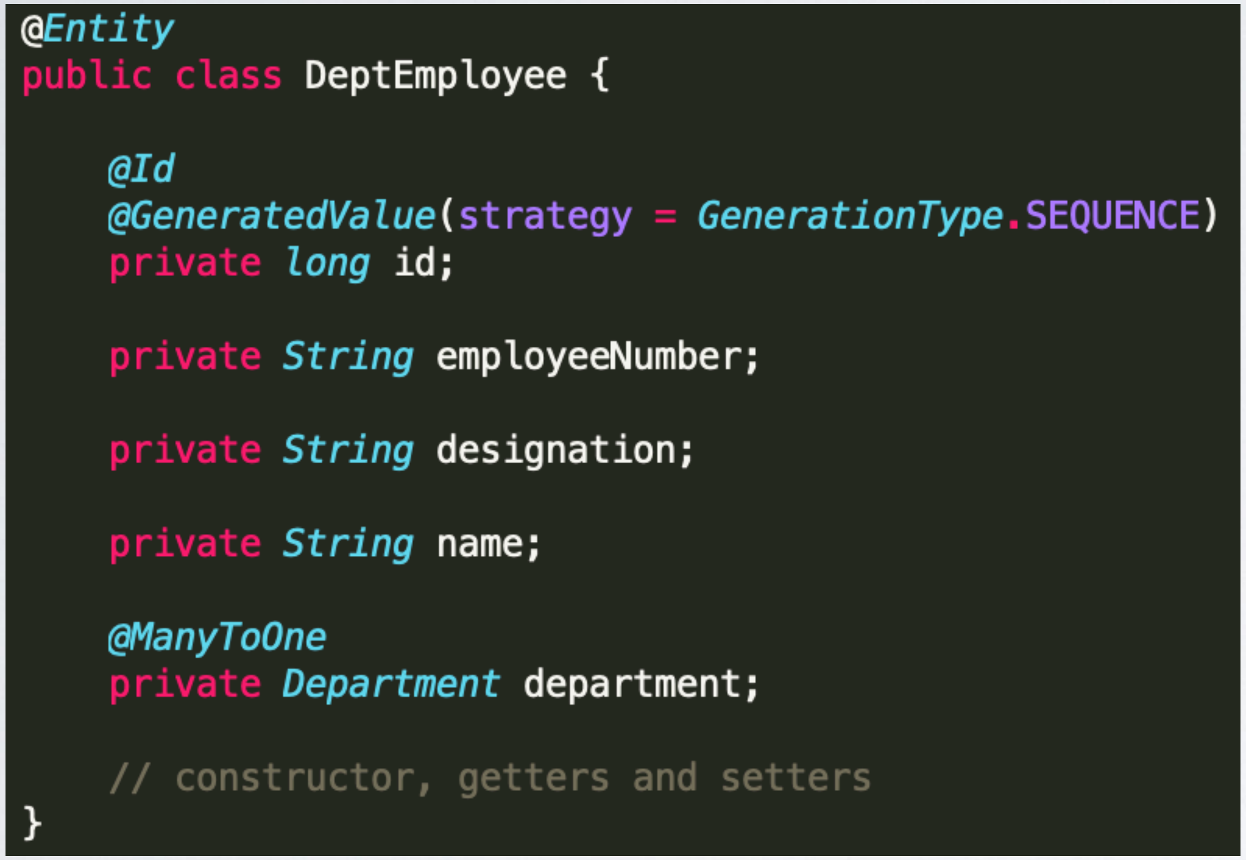


```java
@Entity
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String name;

    @OneToMany(mappedBy="department")
    private List<DeptEmployee> employees;

    public Department(String name) {
        this.name = name;
    }

    // getters and setters
}
```

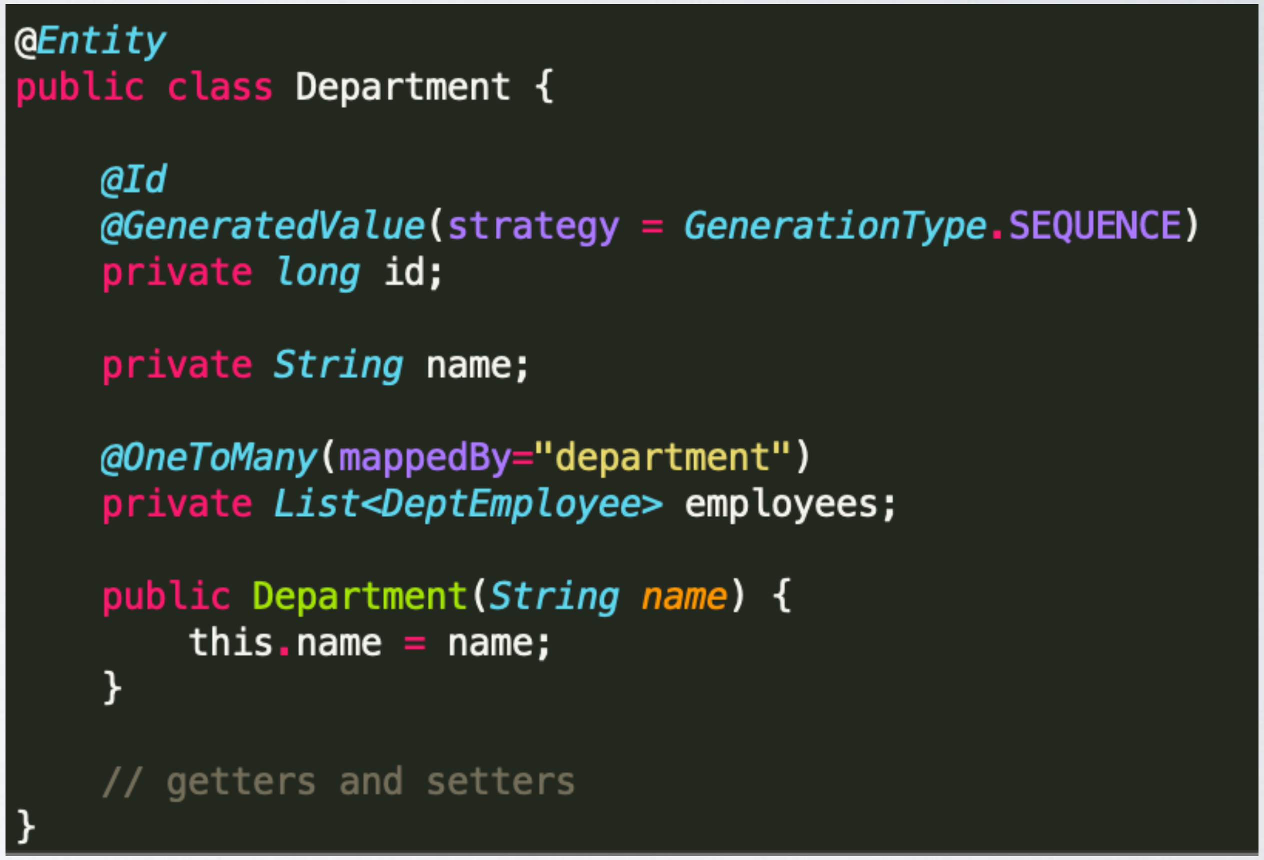


```java
Query<DeptEmployee> query = session.createQuery("from DeptEmployee");
List<DeptEmployee> deptEmployees = query.list();
```

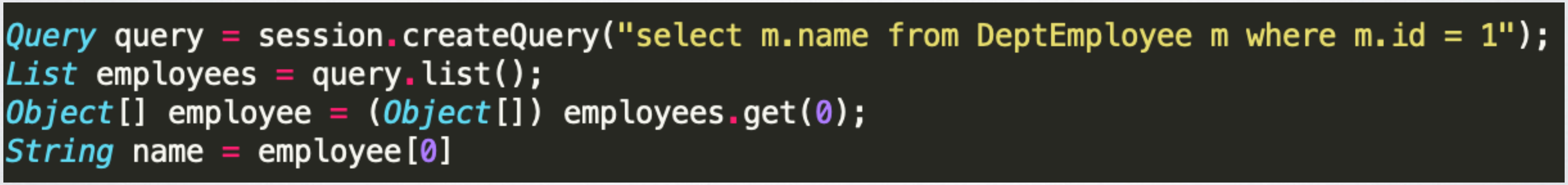
HQL



```java
@Entity
public class DeptEmployee {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String employeeNumber;

    private String designation;

    private String name;

    @ManyToOne
    private Department department;

    // constructor, getters and setters
}
```



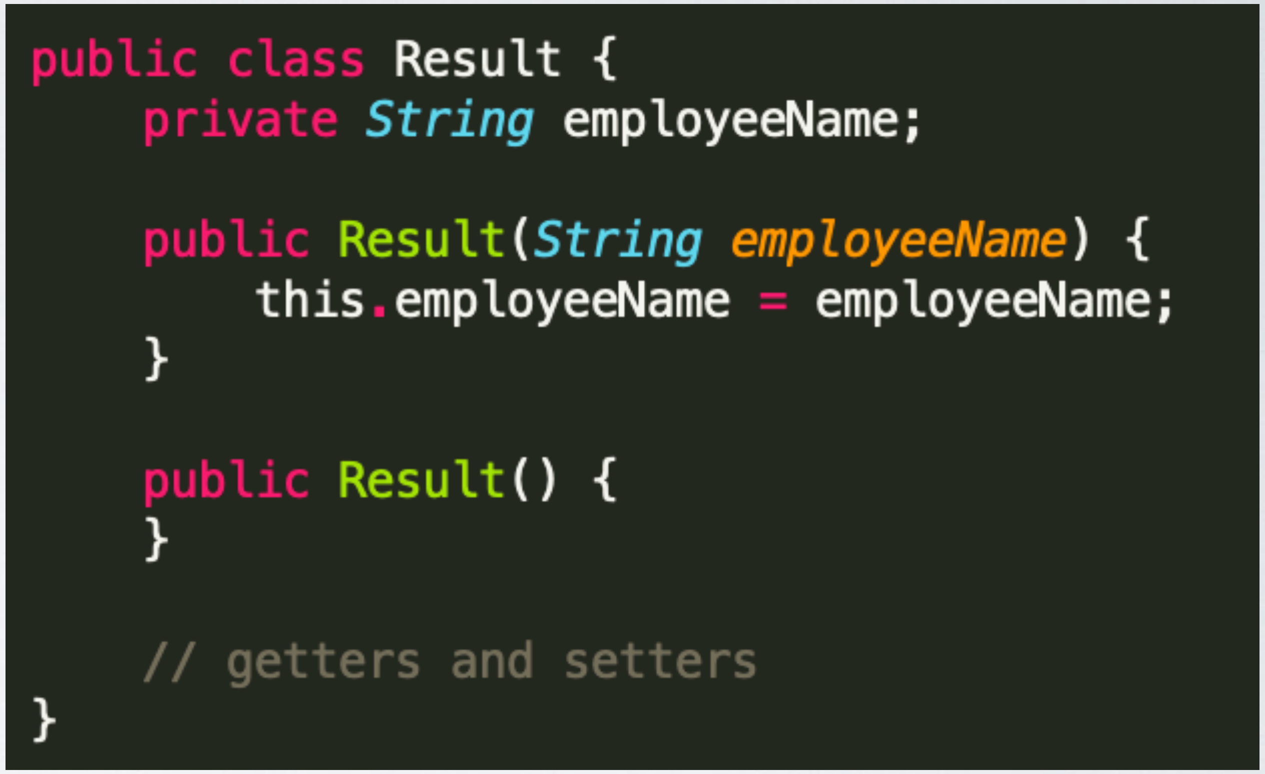
```java
@Entity
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String name;

    @OneToMany(mappedBy="department")
    private List<DeptEmployee> employees;

    public Department(String name) {
        this.name = name;
    }

    // getters and setters
}
```



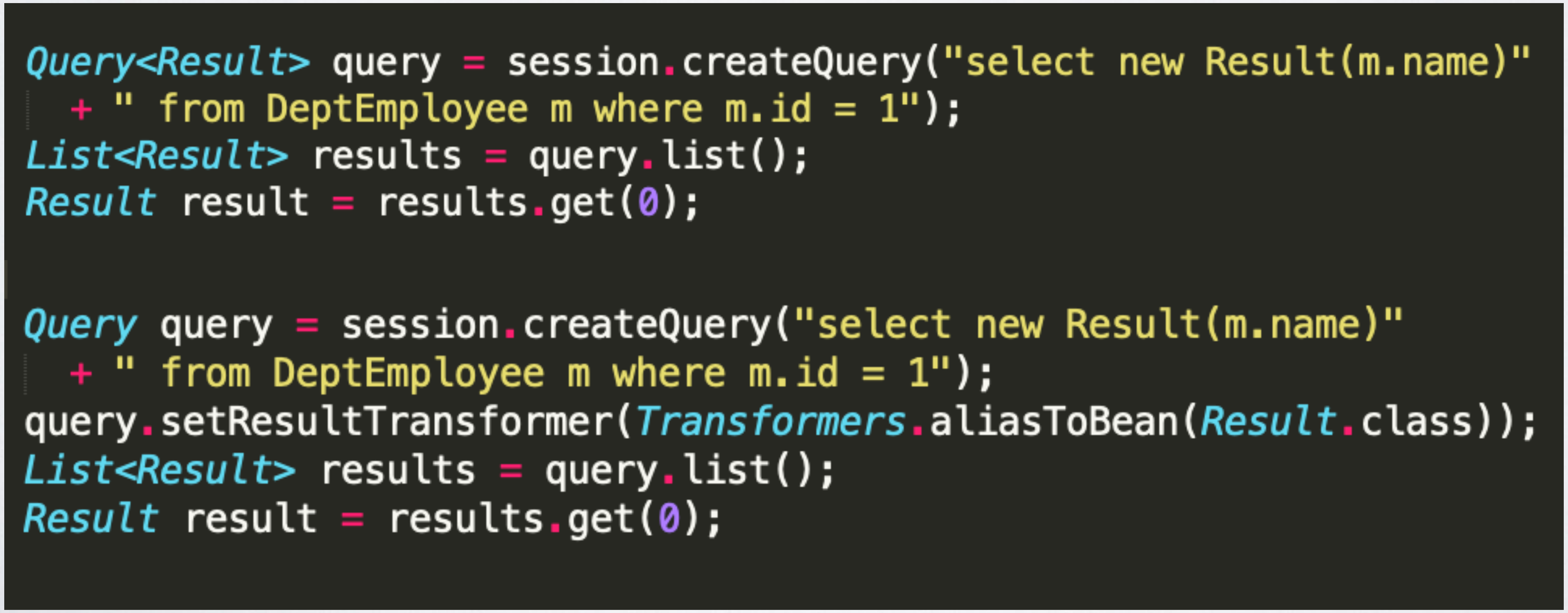
```java
Query query = session.createQuery("select m.name from DeptEmployee m where m.id = 1");
List employees = query.list();
Object[] employee = (Object[]) employees.get(0);
String name = employee[0]
```

HQL

- Custom Query Result
  - Using a Constructor in HQL
  - Using a ResultTransformer



```java
public class Result {
    private String employeeName;

    public Result(String employeeName) {
        this.employeeName = employeeName;
    }

    public Result() {
    }

    // getters and setters
}
```



```java
Query<Result> query = session.createQuery("select new Result(m.name)"
  + " from DeptEmployee m where m.id = 1");
List<Result> results = query.list();
Result result = results.get(0);


Query query = session.createQuery("select new Result(m.name)"
  + " from DeptEmployee m where m.id = 1");
query.setResultTransformer(Transformers.aliasToBean(Result.class));
List<Result> results = query.list();
Result result = results.get(0);
```

ANY QUESTIONS?