## Java Persistence Query Language (JPQL)

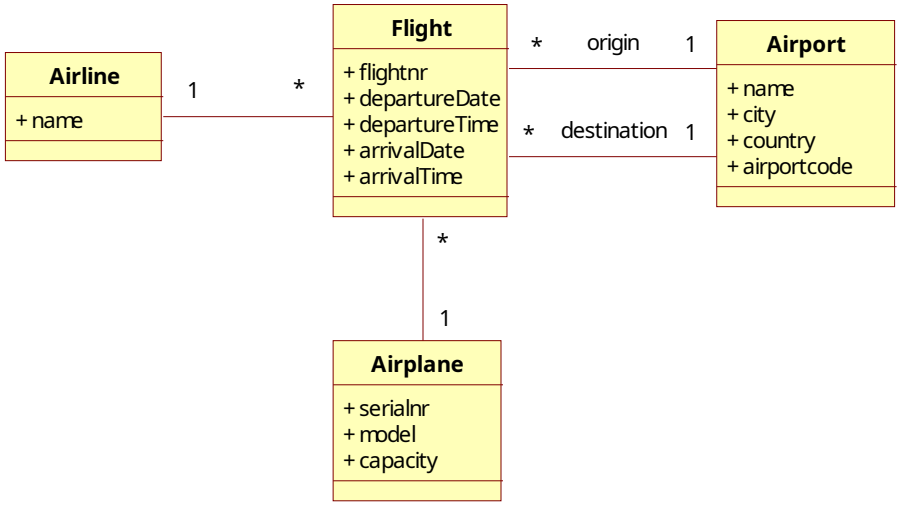
### Exercise JPQL.1

### *The Setup:*

In this exercise you will write several JPQL queries against a given data set. Start this exercise by downloading the provided **W2D3-JPQL** project, and adding the Hibernate, MySQL and log4j dependencies to it.

### *The Application:*

The provided application uses the following flight scheduling problem domain.




Currently the application simply retrieves and prints all the flight info stored in the database. Running the application should produce output that contains something like:

```
Flight:  Departs:                       Arrives:
NW 36    Detroit       8/6/09  7:10 PM  Amsterdam     8/7/09  9:00 AM
NW 96    Tokyo         8/6/09  3:05 PM  Detroit       8/7/09  1:45 PM
QF 12    Los Angeles   8/5/09 10:30 PM  Sydney        8/7/09  6:15 AM
QF 21    Sydney        8/6/09  9:55 PM  Tokyo         8/7/09  6:55 AM
UA 944   Chicago       8/6/09  2:30 PM  Frankfurt     8/7/09  5:45 AM
UA 934   Los Angeles   8/6/09 12:59 PM  London        8/7/09  7:30 AM
NW 8445  Amsterdam     8/7/09  7:15 AM  London        8/7/09  7:40 AM
NW 1689  Detroit       8/7/09 12:05 PM  Chicago       8/7/09 12:21 PM
QF 3101  Los Angeles   8/7/09  3:00 PM  New York      8/7/09 11:39 PM
QF 4022  Tokyo         8/7/09 11:05 AM  Singapore     8/7/09  5:15 PM
UA 941   Frankfurt     8/7/09 12:45 PM  Chicago       8/7/09  2:53 PM
UA 4842  London        8/7/09  8:10 AM  Amsterdam     8/7/09 10:38 AM
```

Note, the data for this exercise is loaded from import.sql inside the resources directory. If hibernate dectects an import.sql file at that location it will always run it on startup.

### *The Exercise:*

Look through the application code, and once you feel comfortable with what it does update the JPQL queries in **Application.java** to retrieve the following data:

a) All flights leaving the USA with a capacity > 500

b) All airlines that use A380 (model) airplanes. Note: while this query can be done using only implicit joins it's important to also practice using explicit joins.

c) All fights using 747 planes that don't belong to 'Star Alliance'

d) All flights leaving before 12pm on 08/07/2009

Hint: you cannot use an **implicit join** to traverse a One-To-Many relationship, and when you use an explicit join you should also use select in your query.

Hint: just like most SQL dialects JPQL expects dates to be formatted according to the ISO 8601 standard which is `'yyyy-mm-dd'`, not the US mm/dd/yyyy format. Times should be formatted in the 24 hour `'hh:mm:ss'` format.

# Hibernate Optimization (HOP)

## Exercise HOP.1 – Data Fetching

### *The Setup:*

In this exercise we will use System.nanotime() to check how long it takes for MySQL and Hibernate to retrieve the same dataset with different fetching strategies.

Start this exercise by downloading the project from **W2D3-Optimization** and add the Hibernate dependencies to the pom.xml.

### *The Application:*

The application has a Populate.java file that will insert 100,000 owner objects, each with 10 associated pet objects into the database. Run it once (will take a while).

Then change line 23 of the persistence.xml file to have the value of “none” instead of “drop-and-create”. This will stop the tables from being re-created every time and keeping you from having to recreate all the data.

Then run App.java, which will create an N+1 and tell you how long it took.

### *The Exercise:*

Consider what the application does, and write down which strategy you think will perform best under these circumstances. To get a more accurate time you should probably run each test 3 times and take the average, but once is okay to get an idea.

a) Add the @LazyCollection with option EXTRA to the association and run App again.

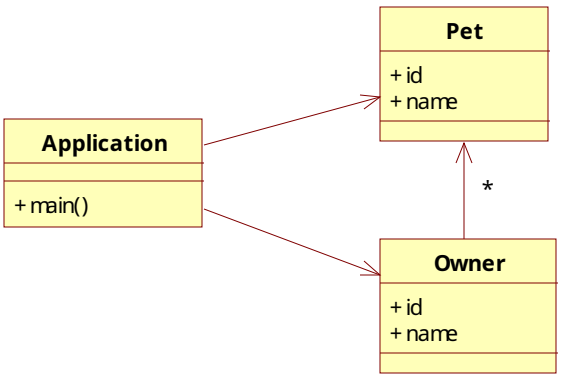
b) Remove the @LazyCollection, and modify the mapping for **Owner.java** to use **batch fetching**, batch size 10. Also check the time when using sizes 5 and 50.

c) Modify the mapping to use the **sub-select** strategy instead of batch fetching.

d) Remove the sub-select strategy and use a **join fetch query** in App.java to retrieve everything. Also check the difference between using a named query, or just a query directly in code.

e) Lastly modify the application to use an Entity Graph instead of a join fetch.

Check to see if the strategy you thought would perform best was indeed the best for this situation. Remember, just because a strategy performed well under these circumstances does not necessarily mean it will perform well under other circumstances.



## Exercise HOP.2 – Removing N+1

### *The Setup:*

In this exercise we will add stratgies to help mitigate the N+1 problems from the JPQL (flights) exercise. Start by making a copy of your solution for **W2D3-JPQL** and calling it **W2D3-HOP2.**

Then set the show_sql and format_sql options to true in the persistence.xml file.

### *The Exercise:*

Use optimization techniques to get rid of the N+1 problems in queries a) c) and d). (Query b) does not have an N+1 problem).