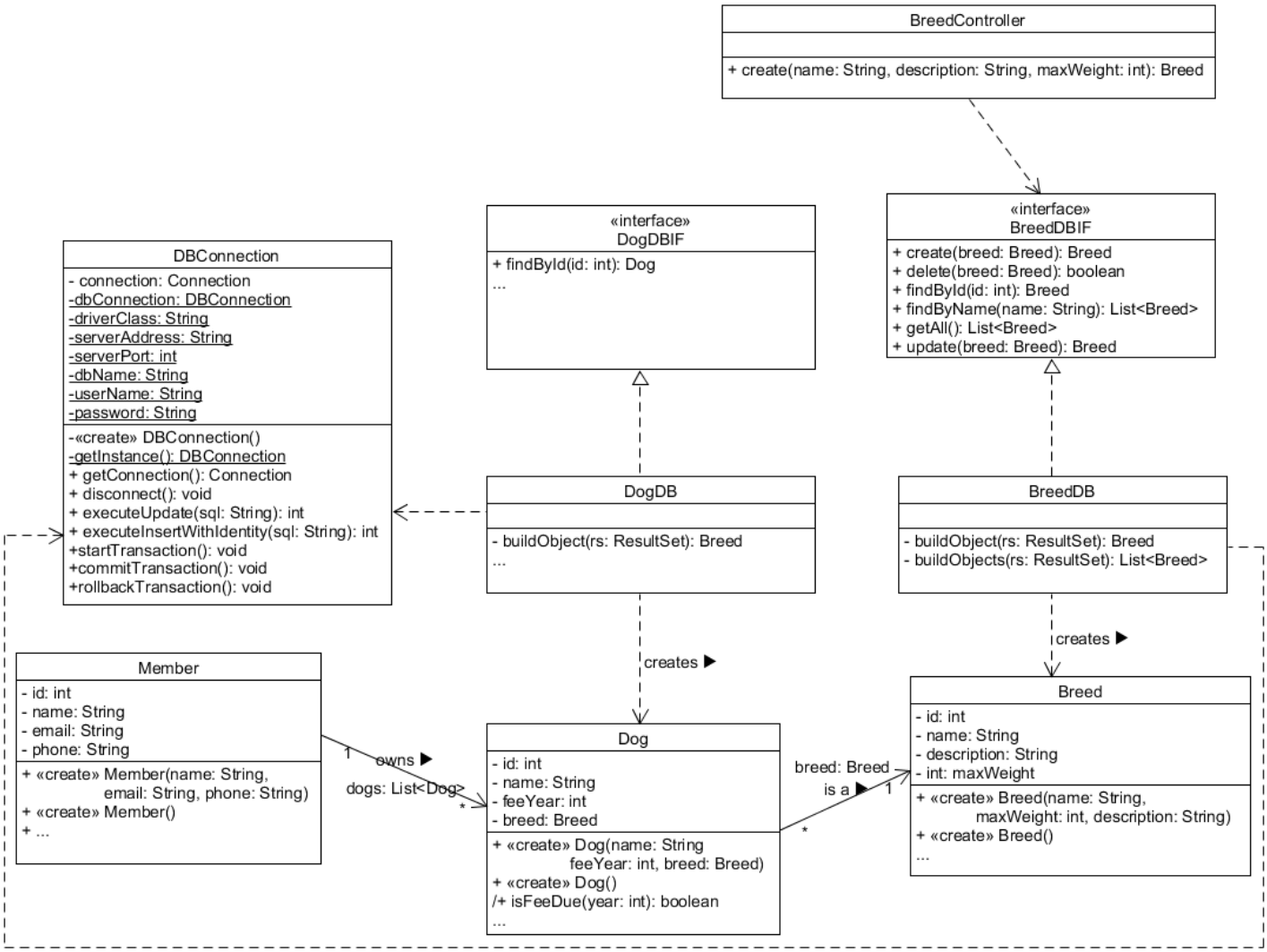
FidoFitness Club with Database **v.0.1 Alpha**

The Fido Fitness Club in Woodmere needs a software system to keep track of members, dogs, the breeds of the dogs in the club, and membership fees. Membership fee is payed yearly for each dog. This is why membership fee payed is registered on the dog, not on the owner (see the class diagram). The method **`Dog.isFeeDue():boolean`** returns **`true`** if the feeYear attribute is less than the current year. The **`Member.isFeeDue(): boolean`** method returns **`true`** if any of the member's dogs isFeeDue() method returns true.

During this exercise, you build a software solution using open, three-layered-architecture, and the **DAO** (Data Access Object) pattern (see the slides about this topic).

Converting relational data to objects and back

1. Investigate the project!
    1. Unzip the .zip file
    2. Import the project into Eclipse
        1. Make sure that you have the **JDBC** jar file in your classpath
        2. Make sure that the **JUnit** library is in your classpath (Eclipse can help you with this if you allow it to "fix" the project setup)
    3. The project is incomplete, but some of the database connectivity is in place
    4. To have a feel for how it works, have a look at what the JUnit test cases do
        1. Follow the method calls from the **TestBreedDB** all the way to **DBConnection** and understand how the data from the client (test class or controller class) is sent around the system.



2. Investigate the **test.TestSuite** class.
    1. Notice how the body of the class is empty, and that the functionality is defined in the annontations.
3. Investigate the **test.DBCleanup** class. The static **`cleanDB()`** method is an example on how to reset a ***test database*** (not a production one!!).
    1. This method is called *from* every **`@Before setUp()`** method in the classes of the test suite. That way the database is reset to the same state before each test case is run, and the test cases can be run in random order as intended by the JUnit framework.
4. The method **`test.TestBreedController.testCreateOK()`** method fails when you run the test Suite. Fix the broken method – JUnit will tell you where to find it (after you have run the test suite).

A good way to organize your program is to follow some simple code conventions. You could agree with yourself to always call search methods "**findBy...**" methods. Like **`findById(int id)`**, **`findByName(String name)`**, etc. Let us add CRUD methods to the controllers, following the naming conventions. Add a create, a delete, an update, and few search methods: findById, findByName, findAll (or getAll). These methods should do two things:

a. Convert the passed-in parameters to model layer objects (if relevant)
b. Persist the changes in the database layer (if relevant)

Notice how there already are corresponding methods in the db-layer, you just have to provide the wiring.

Calling DB-classes from the controller

5. Add the
    1. **`delete(int id):boolean`**
    2. **`update(Breed breed):Breed`**
    3. **`findAll():List<Breed>`**
    4. **`findById(int id):Breed`**
    5. **`findByName(String name): List<Breed>`**

    methods to the **BreedController** class.
    2. Wire them to call the corresponding methods on the **BreedDB** class.
    3. Write test cases in the **TestBreedController** class.

Managing the Breed class is relatively easy – we just need to package and unpackage data to and from objects. The only trick is to notice how the auto **generated id** from the database is **retrieved in the same step as the insert happens** (look in the **DBConnection** class).

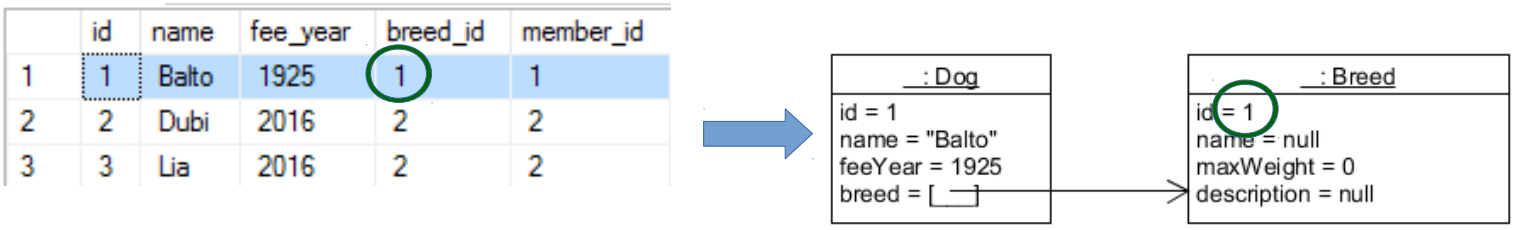
Converting foreign keys to references

Now, we move on to handling foreign keys (database) and references (object oriented program). The task is the same as before, except, now we **convert foreign keys to references** when we build objects and **extract foreign keys from referenced objects** when we write to the database.

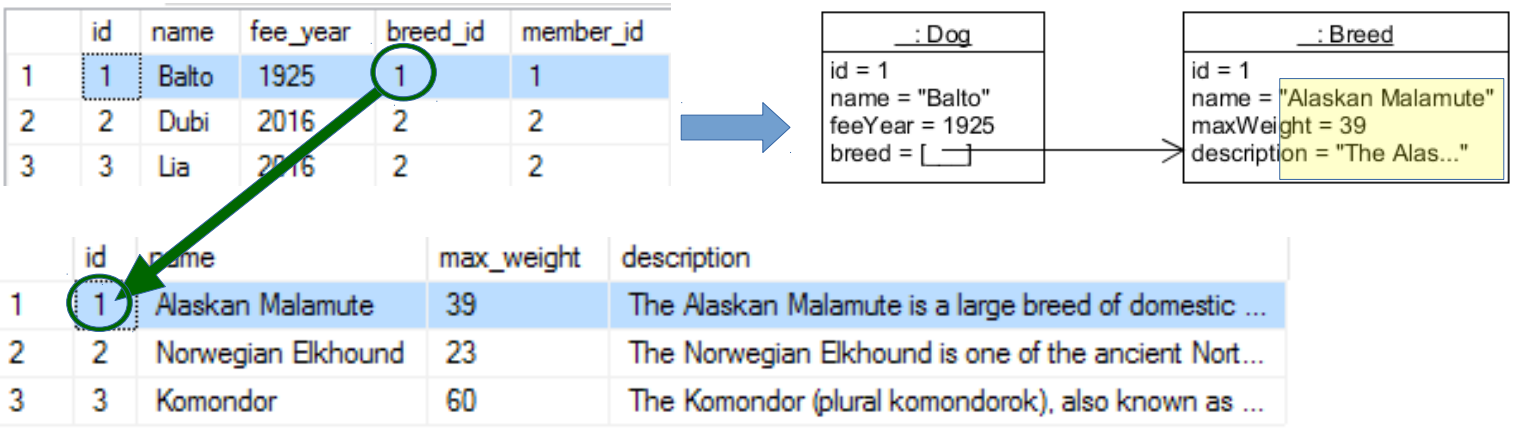
6. Investigate the **`DogDB.findById(int id, boolean retrieveAssociation):Dog`** method.
    1. First, a **Dog** object is created and its attributes are set from the **ResultSet**. This happens in the private (auxiliary) method **`buildObject(ResultSet rs): Breed`**. This method will be reused in all the methods that query the database for Dog objects.

1. The **BREED_ID** from the database is converted to a **Breed** object by creating a "dummy" Breed object with only the **id** set.
2. If the **`retrieveAssociation`** parameter is ***true***, the **Breed** object is fully retrieved by calling **`BreedDB.findById(int id): Breed`** method, and is passed to the **`Dog.setBreed(Breed breed): void`** method, thereby ***replacing*** the dummy object.

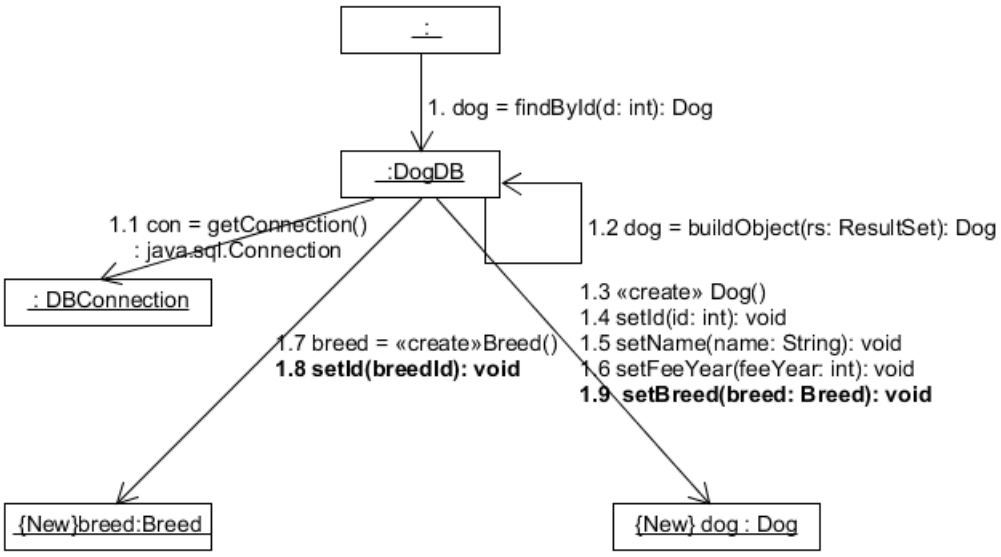
The above is shown in the figure below. Observe how the foreign key **BREED_ID is converted to a reference from the Dog object to the Breed object**. The Breed object is a **"dummy"**, as only its id is set. This is how we convert foreign keys to references. No lookup in the BREED table has taken place.



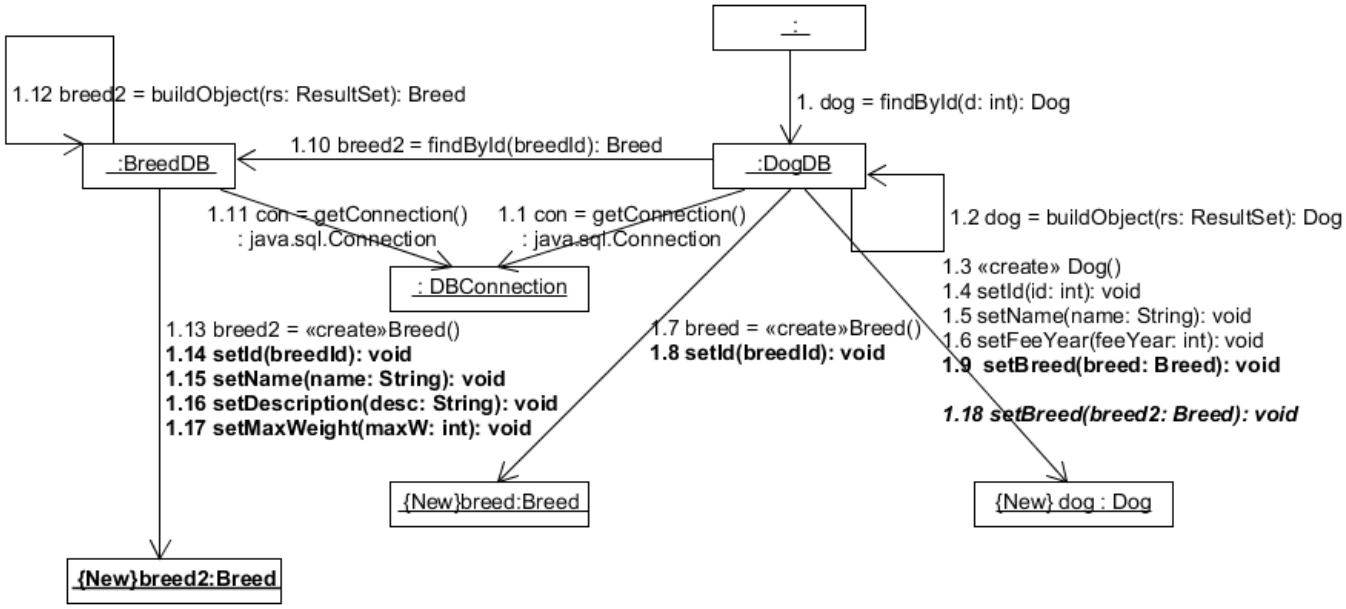
If the retrieveAssociation parameter is true, we want to also build the full Breed object. The below figure shows the logical transformation of the relational model to the objects created. Notice the highlighted (yellow) area in the Breed object! This can be used if your use case requires you to display the details of the referenced variables as well.



Looking at the communication diagram for the first case we see that a Breed object is created in the DogDB class and is provided with only the breed ID.



In contrast, if **`retrieveAssociation`** is ***true***, DogDB collaborates with BreedDB to retrieve the full Breed object and ***replaces the original, "dummy" Breed object:***



Notice that after **1.9, a call to BreedDB** is made in 1.10. BreedDB creates another Breed object and **in 1.18 DogDB replaces the** previously (1.9) set **Breed object**.

7. To gain some experience, you should go ahead and implement the following methods on **DogDB** (look in BreedDB and look at the existing method in DogDB to understand how it is done):
    1. **`create(String name, int feeYear, int memberId, int breedId): Dog`**
    2. **`delete(int id):boolean`**
    3. **`update(Dog dog):Dog`**
    4. **`findAll():List<Dog>`**
    5. **`findById(int id):Dog`**
    6. **`findByName(String name): List<Dog>`**
    7. **`findByMemberId(int memberId): List<Dog>`**
8. Implement the corresponding methods in the **DogController** class (you need to **add that class**)
9. Write some unit tests to verify that everything works

Converting foreign keys to references when the reference goes in the opposite direction of the foreign key

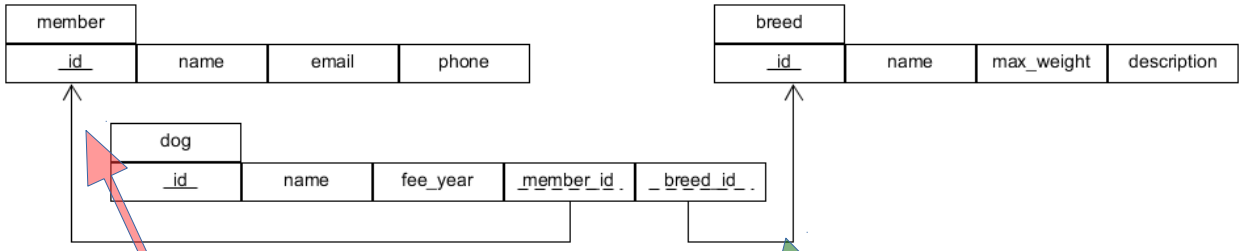
Until now, we have covered **converting a single table without foreign keys** (breed), then we converted a table that has a **foreign key to another table**, and the reference variable in java points in the same way (Dog → Breed).

Notice, however, that the **dog** table has another **foreign key**, which **points to the member table**! The reference variable in Java, on the other hand, points from the **Member** class to the **Dog** class!

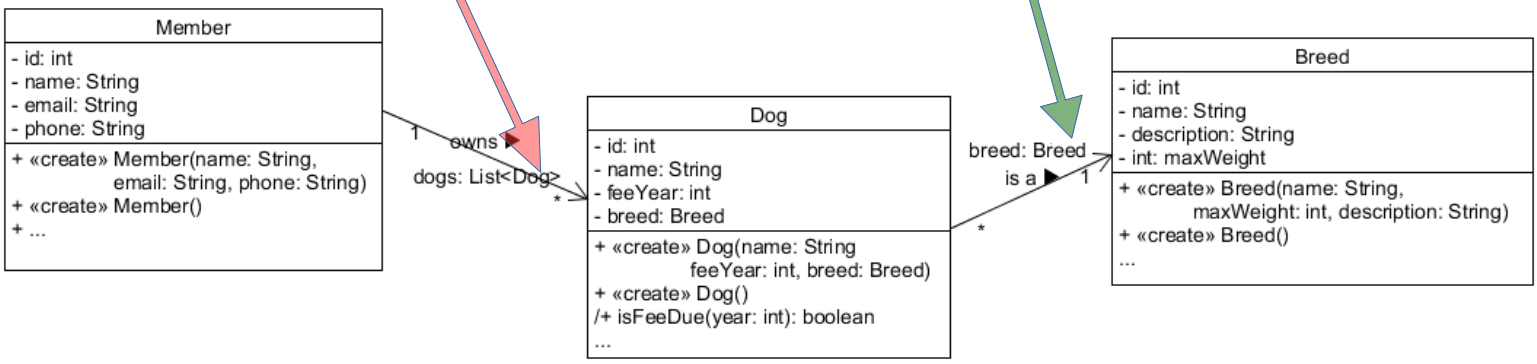
This situation is new, and we cover this conversion in this final section.

To understand the task, we must have a look at the relational model and the class diagram showing the model layer.

The **dog** table has two foreign keys pointing to **member** and **breed**.



The class diagram shows that while the Dog class has a reference, that points in the same direction as the foreign key, ⟺ Dog does not have a reference to Member. The association points in the opposite direction. ⟺ Flipping the direction of the foreign key is not possible in a relational database, as the **1-* relation can only be realized as shown above** – otherwise, we'd have to have a foreign key in the **member** table which contains multiple values, which is not allowed according to the first normal form (1NF) (and not really supported, either).



The simple solution to the above is to reverse the direction of the association in the class diagram (and in the implementation). This would mean that the program design is limited by the chosen persistence model (in this case a relational database). If reversing the association direction is not possible or desirable, you need to handle the multiplicity in the DB-class, which adds complexity to the solution.

The consequence of sticking to the class diagram above is that implementing the MemberDB class is going to be more challenging, as we now must use joins and nested loops to handle the 1-* relationship.

10. Finish the project by supporting Members
    1. Specify the methods for the **MemberDBIF** (add the interface to the project)
    2. Write the **MemberDB** class (it must implement the interface)
        1. Pay attention to the **`findBy…`** methods., they are now somewhat tricky if you have decided to keep the association direction.
        2. Have you decided to flip the association direction, then you must modify the **Dog**, the **DogDBIF**, and the **DogDB** classes, but you have a simpler task here.

3. Write tests. Remember to add the new test class to the test suite and to implement the **`@Before setUp()`** method as shown in the previous examples – the database must be reset before each test run.

Solutions are provided for Exercise 10 are provided for the version where the association goes like this: Member → Dog.

It is not easy when you try the first time. Do ask and involve your group mates and your teacher!

Note:
Some features are too early to implement.
Simply skip those for now (like transactions)