**The FidoFintessClub Member Management System** v.1.1



**The original FFC exercise**

**From the 1st semester**

1. The club needs a program that can be used to register the members and their dog. They want to store the **members**’ **names**, **phone numbers**, and **email addresses**. Also, *each member* has a **dog**. For *each dog*, they want to be able to record the dog’s **name** and the year, the owner had paid a membership fee.

Also, they want to be able to determine for each member whether he had paid the fee for his dog, and whether he currently has a dog. For this, you must implement two methods: **isFeeDue(thisYear:int):boolean** and **hasDog():bool**ean. Also, implement the **describe():void** method that prints the member’s name, phone and email.  
You can use today’s slides as a starting point.

* 1. Draw a class diagram showing all the fields and methods you must implement. If you have already installed UMLet, use it, or just use pen and paper.
  2. Implement the two classes Dog and Member.
  3. Test it creating three members: Joe, Jane and Jean. Joe and Jane have dogs, but Jean has just lost his dog. Fill in all the fields. For each of them, call the isFeeDue(), the hasDog() and the describe() methods.
  4. Do you get an exception in some cases? Fix your code so it doesn’t crash. You may need to check for null pointers before calling a method on your instance variables (fields).

1. We want the dog’s name to be shown after the member’s name, phone, and email in his or her description.
   1. Extend your **describe():void** method to include the dog’s name in the member’s description. If the member has no dog, the method should print “no dog” instead of crashing (with a NullPointerException).
   2. Test it with Joe, Jane, and Jean.

**ArrayList & Iteration Over a Collection**

1. The club has realized that some of the members have – surprise – more than one dog. This should be addressed in the program. For now, they think that 3-5 dogs per member is enough, but they don’t want to exclude the possibility that in the future, some members will have many more dogs – some kennel owners have expressed interest in joining the club.
   1. Re-draw you class diagram to reflect this. This means changing the multiplicity of the association between Member and Dog.
   2. Replace the **setDog**() method with **addDog(dog:Dog):void** in the class diagram that takes a Dog object as a parameter.
   3. Replace the getDog() method with **getDogs():ArrayList<Dog>** in the class diagram. Implement the new class diagram in your program. Hint: you may want to use ArrayLists to hold a number of dogs in each member object.
   4. Now that the reference is a List of Dog objects, we can’t simply set the reference to the Dog class to **null** to indicate that the owner no longer has a dog. Instead, we must be able to remove a Dog from the list. To support this, add the **removeDog(dog:Dog):void t**o the Member class.
   5. Update the isFeeDue() method to return true if the owner hasn’t paid the fee for any of his or her dogs.
   6. Write a method that returns the number of dogs that the owner hasn’t paid a fee for. Update your class diagram.
   7. Write a method that returns (an ArrayList of) only those dogs, the owner hasn’t paid a fee for.
2. As the club grows, they want to have an idea of what breeds the dogs are.
   1. Update your class diagram to reflect the solution to this requirement. Consider using a new class to represent the dogs’ breeds.
   2. Implement your changes from the class diagram!