Lab 10 C212 FA22

Last modified 2022-11-02 11:59 PM

Summary

The goals of this lab are as follows:

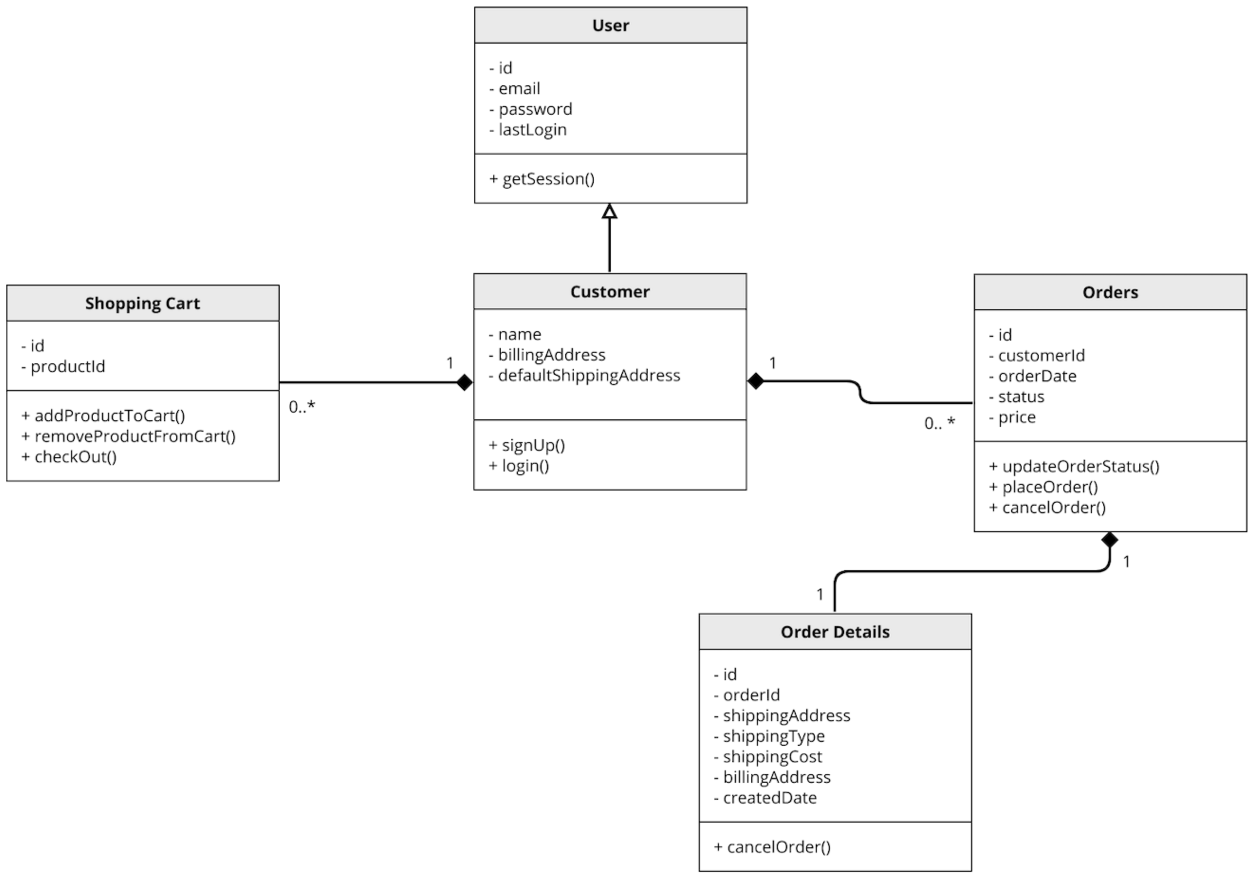
- Part A - Reading UML Class Diagrams
    - A traditional one
    - An IntelliJ-generated one
- Part B - Implementing a class hierarchy from a UML Class Diagram
    - Our lovely Animals
- Part C - Making a PetDaycare where Animals meet and greet
    - But something seems wrong…
- Part D - Letting our Chihuahuas tell us how they really feel
    - Casting Animals to the appropriate subclasses

Part A - Reading UML Class Diagrams

UML stands for Unified Modeling Language, and it is a design convention for many types of diagrams to visually represent relationships in an object-oriented system. Despite being named "Unified" Modeling Language, the actual specifics of a UML diagram can vary based on the style conventions of a specific organization. In particular, we are interested in the UML Class Diagrams, which show class hierarchies and members. We'll look at two styles of UML Class Diagrams and explain how they're read.

Be warned, they are pretty boring to learn about, but you'll probably come across them again in your CS career, so it's important to at least know the basics.

Here's an example of one traditional style of a UML Class Diagram. It's hard to make one diagram that uses every type of symbol and is easy to understand for beginners, but this will give us something to reference and expand upon:



You may need to zoom in to see parts of it, but all these symbols do matter, so let's break them down:

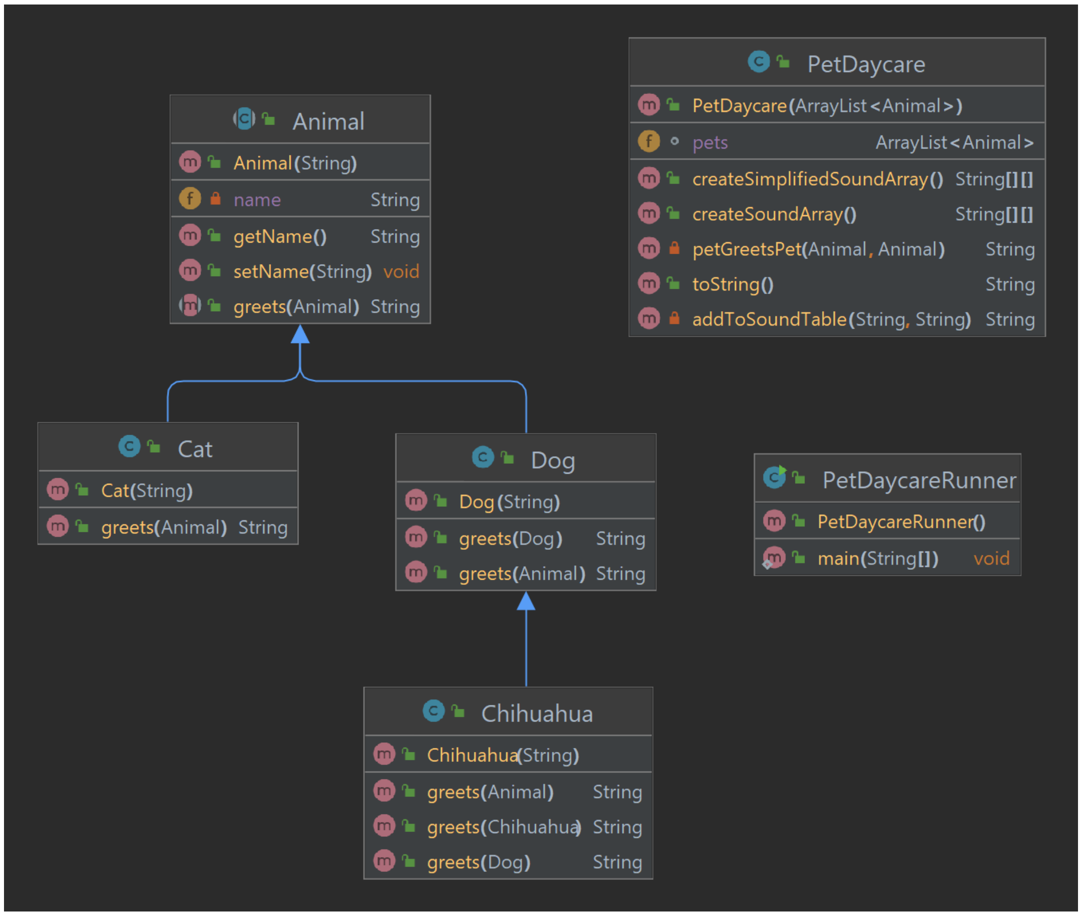
- Each class is one box. In this style, each class box has three sections:
  - The name of the class (in PascalCase, just like our Java classes)
  - The **fields**, a.k.a. variables (in camelCase, like our variable names)
  - The methods (also in camelCase, also like our method names)
- Every **member** (which is a term meaning a class's fields and methods) has either a - or + symbol before it. This denotes the **access modifier**. - means private, and + means public. Protected is rarer, and usually gets the symbol #.
- There are none here, but static members are underlined.
- There are none here, but **abstract** classes and abstract members are italicized. Abstract classes and **interfaces** also usually have <<abstract>> and <<interface>> below their name, too.
- The arrow from Customer to User shows **inheritance**, so a Customer "is-a" User.
- One important and somewhat contentious convention choice is whether methods show their parameters and their types. In this case, the methods either don't

have parameters, or the style used for this diagram excludes them. Arguably, it is worth the clutter to include these parameters and their types for each method.

- The diamond arrows represent **aggregation**. This is the "has-a" relationship, just like the relationship between Book and Author in A6.
    - The numbers floating around them are their cardinalities in the relationship, basically meaning how many of either entity exist in that relationship. A good way of thinking about this is the difference between Book and Author vs BetterBook and Author in A6. A Book has 1 author, while a BetterBook has at least one author but potentially arbitrarily many. In this case, BetterBook would get 1..* as its cardinality, the star meaning an arbitrary amount. The most common cardinalities are 1:1, 1:0..*, and 1:1..*. Don't worry about cardinalities too much for our purposes.

However, a lot of these rules will vary, again, based on the style conventions of the organization making the diagram.

Here's another UML Class Diagram, this time one that was generated using IntelliJ:



Notice a few differences here:

- Classes have a C, methods have an M, and fields have an F.

- Abstract classes and methods have gray sides.
- The public access modifier is a green unlocked padlock, while private members get an orange locked padlock.
- Our classes here actually have four sections, where the new second section shows the constructor(s) for the class.
- The methods here DO list both the types and number of parameters they take, and the return types are given on the right, too.
- Runnable classes get the play icon over their C, and the main method gets a gray diamond over its M.

If you ever want to, you can generate your own UML Class Diagrams in IntelliJ for the code you write by selecting the files you want shown, right-clicking > Diagrams > Show Diagram…, if you have IntelliJ IDEA Ultimate Edition.

Now that you know everything there is to know about reading UML Class Diagrams, your task is to implement the system shown in this diagram!

Part B - Implementing a class hierarchy from a UML Class Diagram

Referencing the above UML Class Diagram, in Part B, you will implement Animal, Cat, Dog, and Chihuahua. Pay close attention to the diagram! For example, Animal should be abstract.

The most important part about this system is the greets() method for each Animal. Here are some rules regarding how those should work:

- The greets() method in Animal itself is abstract. That means this method does not define the functionality of greets(), but it does mandate that it should be there in each subclass.
- Cats are uninterested in other animals. When a Cat greets any other Animal, it just says "Meow".
- Dogs like to see other animals, but they especially like to see other dogs. When a Dog greets another Animal, it says "Woof". However, if a Dog sees another Dog, it says "Howl!". This is done by **overloading** the greets() method in Dog.
- Chihuahuas are weird little dogs, known for being kind of annoying. When a Chihuahua sees any Animal, its default greeting is "Arf". When a Chihuahua sees another Dog, however, it says "Arf Arf!" because it wants to act tough. But when a Chihuahua greets another Chihuahua, it says "¿Cómo estás?" – this is a little known fact but is actually true if you listen closely.

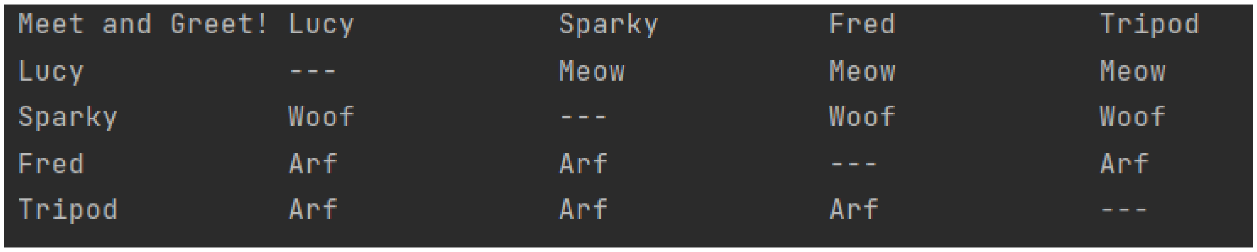
As you write these classes, you are welcome to test them however you wish to ensure things are working as expected, but you aren't required to for this task.

Part C - Making a PetDaycare where Animals meet and greet

For this part, we're working on class PetDaycare, which aggregates our Animals. It takes in its constructor a list of Animals and stores them in an instance variable pets. The most important part of the PetDaycare is the meet and greet, where each pet greets every other pet. To do this, PetDaycare has a few methods, some of which you need to work on for this task (ignore the commented-out ones for this part):

- createSimplifiedSoundArray takes our list of pets and generates a 2D array of sounds that are heard when the pet in row i greets the pet in column j. Refer to the code output below to help visualize what this should look like. Your job is to make a couple of easy edits to get this method working. You'll see later why it's called "simplified".
- addToSoundTable is used in the next method to help create a visual representation of the sound array. It uses Java's String formatting to add sounds into nice columns. It's done for you, so no need to edit it, unless you want the columns to be wider or narrower.
- The overloaded toString method will format the sound array into a nice table. It takes that sound array and adds some labels to the top and left of the pets' names, and fills in the diagonals with some dashes.

Once you fill in all the …'s in the starter code, you should be able to use PetDaycareRunner to get this output. Remember, you read this as "the pet in row i greets the pet in column j by saying…"



```
Meet and Greet! Lucy           Sparky         Fred           Tripod
Lucy            ---            Meow           Meow           Meow
Sparky          Woof           ---            Woof           Woof
Fred            Arf            Arf            ---            Arf
Tripod          Arf            Arf            Arf            ---
```

But notice that all the hard work we put into different greetings isn't working! In particular, the Chihuahuas seem to be giving the wrong greetings. In the next part we will fix this.

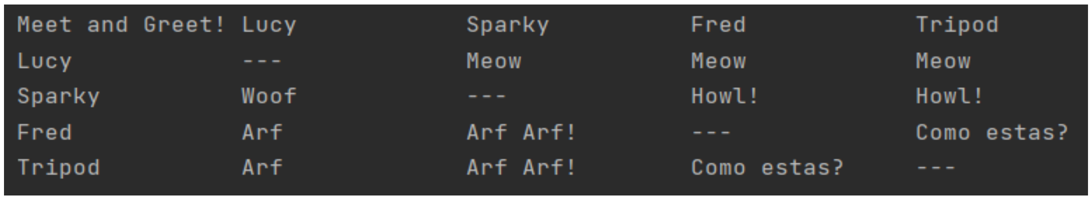
Part D - Letting our Chihuahuas tell us how they really feel

What we're doing by putting all the different specific subtypes of Animal into one list just typed as Animal is an example of **polymorphism** at work. Since Dogs, Cats, and even Chihuahuas are all Animals, we can put them all into one list of type Animal. However, the issue comes about when we want to then actually treat these Animals as their specific subclasses. See, right now a Chihuahua can't seem to tell the difference between a Cat and a Dog and a Chihuahua, and even a Dog doesn't know whether it's looking at another Dog or any other Animal. This is because when we pass in the petBeingGreeted to the petDoingTheGreeting's greet() method, it's just typed as an Animal because that's the type of our list.

We need to convert the petDoingTheGreeting and the petBeingGreeted to the right types, but even before that we need to find out which types those pets are supposed to be! This is where the petGreetsPet method comes into play. The comments explain pretty well what's happening, but as part of this task you do have to fill in the blanks on the comments. You should be able to think about what's being said and fill in the blanks yourself.

The way this code works is that we use casting to convert instances of Animal to their appropriate subclasses. Even though the list treats them as Animals, they are still their usual types. If we knew every Animal in the list was a Dog, we could just cast each one to Dog and then do the greetings, but we don't know, which is why we need to use instanceof. Instanceof is a Java operator that tells us whether or not an instance is an instance of that class anywhere in its inheritance hierarchy. So, a Chihuahua is an instance of Dog and is also an instance of Animal. Notice, though, that we need to check the more specific types first, otherwise we'd just treat Chihuahuas as Dogs if that was the first thing we checked for.

Once you have the blanks filled in in the comments, make the fixes to this method to get it to work, and also do the same things you did in createSimplifiedSoundArray to createSoundArray. To see the results, update the toString method to use createSoundArray instead of createSimplifiedSoundArray, and run PetDaycareRunner. You should get this output, where our pets finally act like their true selves:



```
Meet and Greet! Lucy            Sparky          Fred            Tripod
Lucy            ---             Meow            Meow            Meow
Sparky          Woof            ---             Howl!           Howl!
Fred            Arf             Arf Arf!        ---             Como estas?
Tripod          Arf             Arf Arf!        Como estas?     ---
```