I took notes, as I went. These are my raw notes on my experience and thought process. It also includes what I changed, and why I changed it.

1. Briefly looked at the Repo on GitHub.
    a. Tried to make sense of what was being asked. Didn't really know what to expect at first.
    b. I realized that there were 5 requirements.
    c. I realized that it had to do with aligning pet preferences with a person.

2. I cloned the Repo and opened in VS 2019.
    a. I could help but see that there were already 3 forks. Seems like an exposure, so I chose not to fork it on purpose (yet). I cloned the repo and added it a private repo on my own account, for now.
    b. I realized that there was only 1 project, which had no existing Unit Tests.
    c. I realized that the one existing project was a console application. It was a .NET Core 2.1 app.
    d. I briefly looked at the main program class, and felt like I could run it safely.

3. Run the Application.

    a. I saw that that output had 2 people and a list of possible pets (as good or bad matches).
    b. I opened the main program class, and in order to learn what it was doing, I thought that I might attempt to refactor it into smaller single responsibilities. But wasn't sure that is what purpose of interview exercise was for.

4. Testing & Refactoring Existing Code

    a. At this point, I had to decide between showing what I'd do in concept for real world code (add tests, refactoring) vs. just do the changes required by the interview exercise. I chose to some refactoring, primarily because I thought it would help me ensure I can add the features faster. And it would also demonstrate how I think about coding. Before I changed any code at all, I chose to add a new Unit Test project. Primarily because I wanted to make sure that any refactoring I did, didn't break existing functionality. Since most of the existing code was just models, I realized I would only need to Test the functional program logic.
    b. Added main project reference to Test project. Since Main() returns void, I added 3 test case which simply called the Main() function with different number of arguments. The implicit assertion is that there were no fatal exceptions. In order to do this, I had to change the class and method to Public, as well, I had to remove the ReadLine() call. I plan to revert these changes after the refactor. I felt better refactoring code to be more testable, since I had these first 3 tests, but I could foresee removing them later.
    c. I extracted the Initialization region to its own class (*InitializationManager.cs*). This would separate the initialization routine from the logical portion of the application class (*Program.cs*). This was easy cause I used Resharper. I extracted the decision-making routine ("isGood") to its own class. I chose to name this "MatchManager.cs" cause I thought its single responsibility was to decide if there was a good match or not.
    d. After this refactoring, I ran the existing tests. They all passed.
    e. I change SetupObjects() method to return a tuple of People and Pets. I did this because I think it's cleaner to get the 2 model objects (pets and people) completely initialized and returned by the same class. That also meant I could remove the class level properties from *Program.cs* (cleaner). The biggest benefit, this change would also make the SetupObjects function, which returned nothing, more independently unit testable. Instead of a tuple, I could have used a class, but in this specific case, I think a tuple is a better representation for returning just Pets and People.
    f. Because of the refactoring I did in previous steps, I felt now I had a decent separation of responsibility concerns. I could now add more unit tests for each separate class, without initializing the exact same single class that also did everything else (the program.cs class). The only thing that was left in Program.cs, was the code that called SetupObjects() and wrote the results to the console. I refactored the "string.Format" style to string interpolation style, for console outputs. This made the code line shorter, and a bit more readable. I think SetupObjects() is bad

name, but not quite sure what to name it to just yet. Plus, I'll leave as-is, and keep the method static, since there is no real logic in it.

g. I added Unit Tests for the new InitializationManager.cs, which now had the code to return the tuple containing the array of pets and the array of people. This was easy because it (currently) is only initializing data structure, so no logic testing. But that doesn't mean that we might not screw up the initialization (logic to set values), so I wrote tests against it fairly quickly. I added Unit Test for the new MatchManager. But since the isGood returned a string, I thought I'd refactor it to return a Boolean. That would make the stronger type, and also make it a bit easier to test, and understand what it's returning. It also made the code a little cleaner by removing the "magic strings".
h. I removed the 3 original tests that tested the Main() function. Since everything else downstream was now being tested independently. The only code in the Main() function was looping and writing out to console. I then put the .ReadLine() code back in. I wouldn't have done this earlier, because I wanted to make sure the Unit Tests didn't get hung up on the .Readline() method. At this point, I now had 4 tests which ensures the existing code for determining a match, and Initializing the data objects works as expected. More importantly, it helped me understand the code that was there. I couldn't measure total code coverage, since I was using community edition. After I did this, I suspected I had proven how important I think adding unit tests to existing code bases (and making code more independently testable was), before adding enhancements. :)

NOTE: I chose not to pre-optimize anything; I tend to leave that to last. I only wanted to break apart the code a bit more. And then add tests. At this time, it probably took me 15-20 minutes to do everything above, but it took 3x-4x that to ensure I documented step-by-step what I was doing along the way (in this file). I decided I'd make a single commit.

5. Requirement One

a. Reading the instructions, at its simplest this seems to be to just add Weight to the Pet model, and then add the pre-defined weight values to each pet. Seems easy enough.
b. I chose to use double over decimal, cause if I recall right, it has less precision. And the values provided doesn't require more precision.
c. Added some test to ensure I can pull a correct value out of initialized object. Added some test to ensure I cannot pull a correct weight value out of initialized object for a pet that doesn't exist.
d. Git Committed and Pushed.

6. **Requirement Two**

a. Reading the instruction, at its simplest, this seems to be to just add a preferred or desired Weight (for a pet), per Person. The weights are a range, with distinction (meaning no overlapping ranges).
b. Since I have the weight of the pet already, I feel like I should be able to extend the Pet object to have a *WeightCategory* that determined by its own weight. Because I want to leave the PetModel as a pure data structure (no logic), I will use an extension method for now. I might see other reasons to not use an extension method after doing the other requirements. I could've probably used a partial class as well. I am not sure I needed to do this just to meet requirement #2, but I suspect I will for requirement #5.
c. Added "PreferredWeightCategory" to the Person type. This is the enum that stores the friendly description of the weight categories.
d. Of course, added tests.
e. Git Committed and Pushed.

7. **Requirement Three**

a. I am not sure what exercise instructions really mean. Does each property simply needs to be virtual (aka overridable)? For what purpose? The word "overrides" has some ambiguity.
b. Or does it mean that the preference per Person should be dynamic to be overridden at run time? Maybe these are the same thing? There's only 1 line of instruction, to makes me think (for the purposes of this exercise), they just want to see what I come up with?

c. After reading requirement #4 and #5 (ahead), it makes me think that they might be looking for me to add a preference order that is not limited to just one preference. I think. So, when I get to #4 or #5, I think I'll add something called a *PreferenceOrder* on the Person object. Each person can then define their order of preference.
d. But for the purity of meeting requirement #3, I'll probably just make each property they listed as virtual. Not sure there's any tests I could do. This requirement doesn't really say to override anything. So, seems like testing requirement #3 would be just testing that C# works. So, I won't add any tests for this requirement.
e. I am not sure I did this requirement right. I'm going to assume that once I start making sense of requirement #5, I may need to make adjustments here.
f. Git Committed and Pushed.

8. **Requirement Four**

a. This is requirement is also lacking good description. I will try my best to figure out what is needed.
b. When the requirement says "a person shall have unlimited overrides" I am going to make a couple assumptions. First, that overrides refer to preference overrides. And that if they are unlimited, I'd need to use a List or Array of them. I'm probably going to add a "PreferenceOverrides" type, and a List<> of them on the Person object. Since requirement #3 identified 3 fields as overridable, for now I'm going to add those properties (as non virtual) to a data structure for an "override" – which I'll call PreferenceOverride.cs.
c. I'm starting to think that maybe an "overrides" on requirement #3, really didn't mean the *virtual* keyword. But I'm not yet sure.
d. While I was able to introduce the Preference Overrides, to this point I was not provided any data per person to add for overrides. So since I didn't add any new logic (or setting of values), I will not add any Unit Tests here either.
e. Git Committed and Pushed.

9. **Requirement Five**

I think this is really 2 parts.

a. First part, ensure a preference order, based on a specific property order. This preference order sounds global, meaning, it's the behavior for all pet matches. I don't think this first part if about eliminating preferences. I plan to order them in preference listed on the requirement. I created another extension method, for the single purpose of doing the ordering. I decided to leave the "good" and "bad" nomenclature, and not suggest a best, better, etc., since I've already ordered them in preference. I added a couple Unit Tests to ensure the re-ordering worked as expected.
b. Second part, now the "overrides" thing is starting to sound more like a hard "objection" to a specific preference. Meaning, if someone has an override, it means that they DON'T want that preference. I think this because the instruction for requirement 5 says " if Person A was opposed to dogs, we would want to be able to add an override for PetType.Dog so that this would no longer be a good match." I think I'm going to call the Overrides something like "PreferenceOverrides" to be clearer about its intent. I also think I'm going to add an Overrides for each type. I added a couple unit tests to ensure that someone with a cat dislike (a cat override) would not get a "good" return for a cat. You'll see this in the difference between the commit for requirement 4, and this for requirement 5.
c. Git Commit and Push.

Done. I'm still not entirely sure the *virtual* keyword was the right thing to do for requirement 3. I think override in this case, meant a functional concept, not necessary a programmatic one. But I think I got the gist of the exercise, which was to enhance the alignment between a Person and possible Pet selection, with the ability to add hard objections called "overrides".

I plan to upload this document as my final commit.