1 Solutions

1. Provide definitions for the following:

   (a) **Abstraction**: The idea of one entity providing an interface to another entity. This is really important in OOP because our paradigm relies on letting an object handle a task, or responsibility, and if other entities accesses that information, then they may be able to change it in an undesired manner, which would make it difficult to go back and change the code later. Consider a situation where we're making a video game, say Halo. An example of good abstraction would be only allowing the Grunt(bad guys) object to change its own health, and instead giving the Master Chief and Arbiter (good guys) an interface to do damage. This is important because if we decide to make the grunt weaker, then we would have to change both how much damage Master Chief and the Arbiter do. However, if we use good abstraction, then we'll only have to change the damage done in the the Grunt object.

   (b) **Encapsulation**: Encapsulation is the act of actually hiding the implementation details of an object from another entity, which forces the program to use good abstraction. The most common implementation in OOP is making variables of an object private. This really important to OOP as mentioned earlier, because it forces abstraction. In our previous example, if the Grunt's health variable is private, then there is no chance that Master Chief and Arbiter will change it. Instead, they will be forced to use the public interface.

   (c) **Cohesion**: Refers to the "clarity" of an object, or how focused its purpose it. This in turn provides insight into what the purpose of the object is. Therefore we want our object to have high cohesion so that our objects are very well-defined. Going with the Halo theme, if we have a Grunt object that has methods `takeDamage()`, `callCovenantShip()`, `renderMasterChief()`, and `handleUserInput()`, that is an exaggerated example of atrocious cohesion, because the routines in the object are barely related, if at all. However, it would be much better if the Grunt object has the private variables `health`, `location`, and `currentAction`, and the methods `takeDamage()`, `throwGrenade()`, and `disappearAfterDeath()`, as these all clearly pertain to things that the Grunt is/does.

   (d) **Coupling**: The complement to cohesion, cohesion measures the strenth in connection between two objects. However, here we want low cohesion so that two objects are connected in as few points as possible. If two or more objects are always calling and depending on each other, then that is a good sign that they

have very low cohesion. Also, a major change in one object is also very likely to create major changes in the objects that depend on it, which is a hassle. For example, if we use the same methods from cohesion, it's very likely that if the Grunt object is handling user input, then the other class(es) that are handling user input are going to depend on that. Then if Microsoft decides to create a new controller, then they'll update the handling class accordingly, but then also have to update the Grunt class for some reason.

2. First of all, the level of abstraction on Slide 6 of the textbook is bullet points, so that's what I'll be using. I'm assuming that some HR employee passes the hours worked into the database earlier in the week, so that when it's queried, it shows how much each employee worked. I'm also assuming a very high level, so I'm not going to mention any error handling or edge cases. So here are the steps, each of which would approximately be a function:

   - Earlier in the week, an employee sumbits a list of hours to an interface
   - Then the interface queries the database and updates all empoyees' hours
   - Later, query database for relevant employees. Return a list of lists
   - Take list and calculate pay for each employee. Return this along with bank accounts
   - Iterate through each of the employees' pay and bank account info
   - Pass into a function that pays a certain amount through the company bank

3. The first and most important thing is an Employee class. Earlier in the week, the HR employee would enter everything into the UI, which would then interface with the Employee object, which would then interface with the QueryDatabase object to update their pay. Then you could iterate through every employee and have them handle their own business. They would talk to an interface on a QueryDatabase class, then calculate their own pay, and input their information to a MakePayment class. Notice how in this paradigm, if we change how we connect to the database, then we can fix the corresponding code in exactly one place, whereas there were two functions accessing the database in the first, functional example.