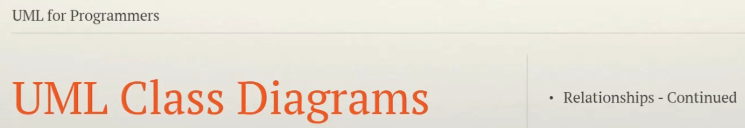
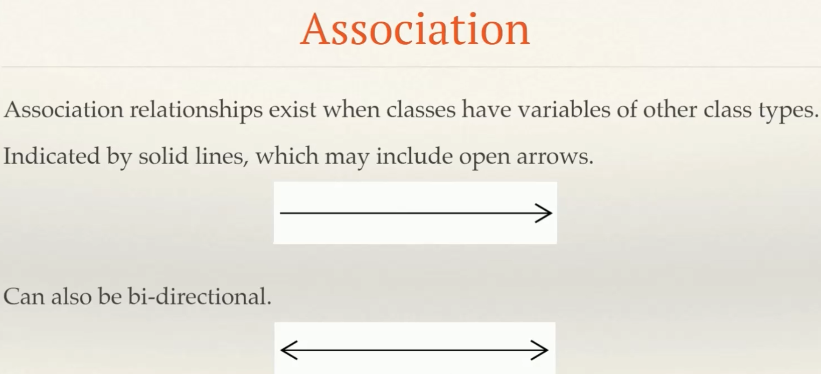
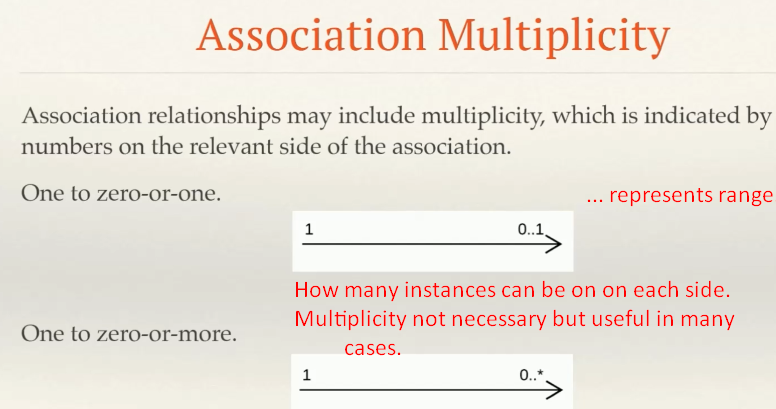
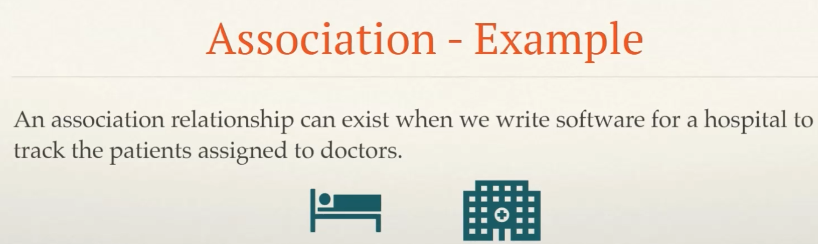
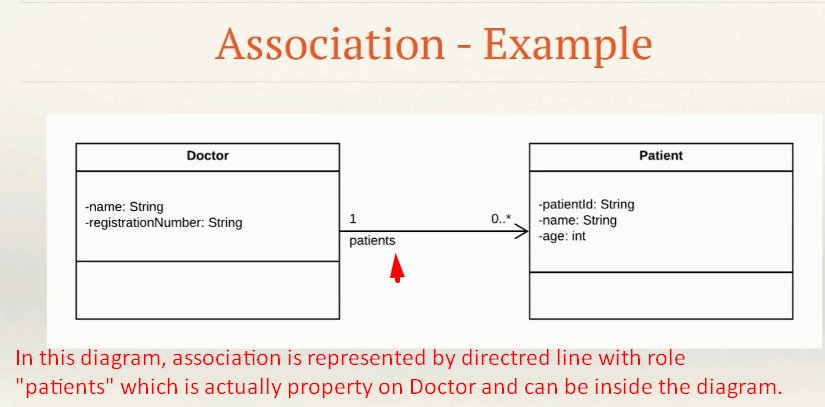
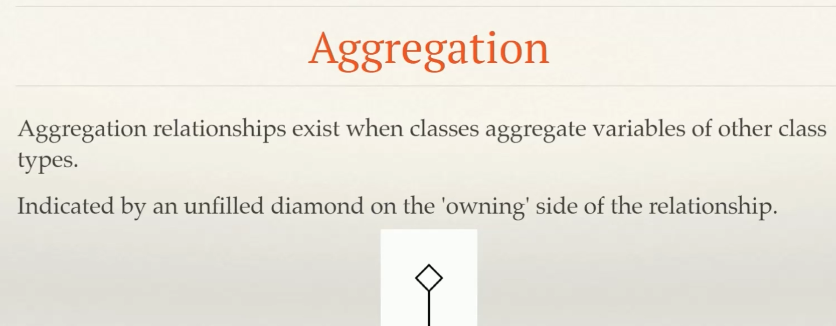
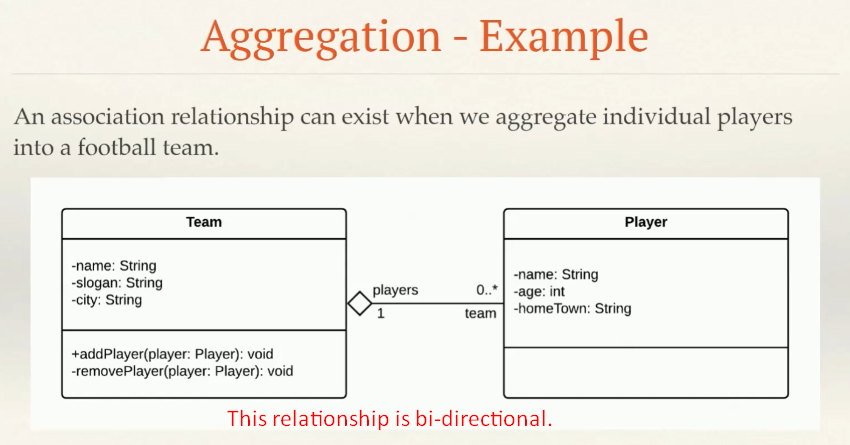
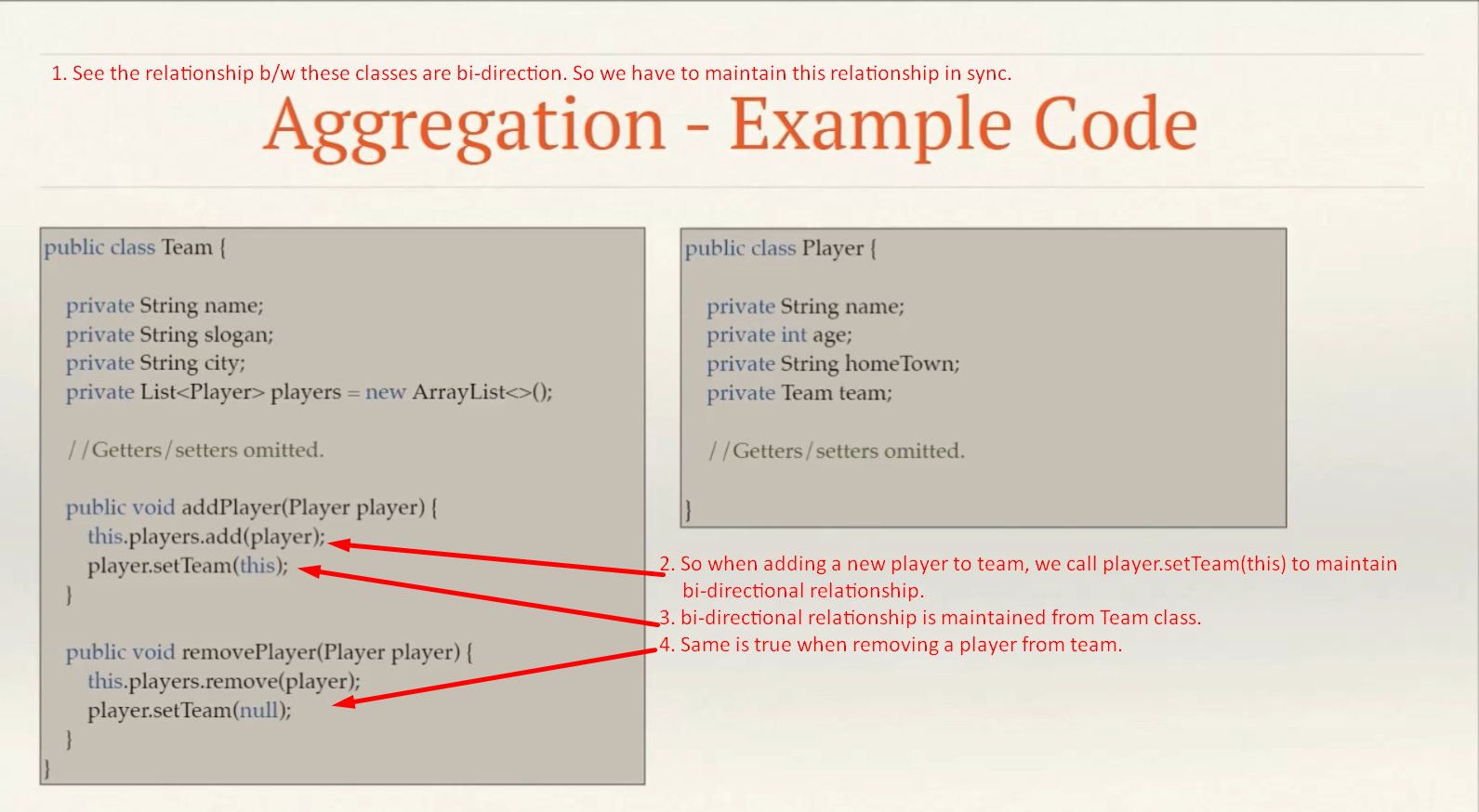
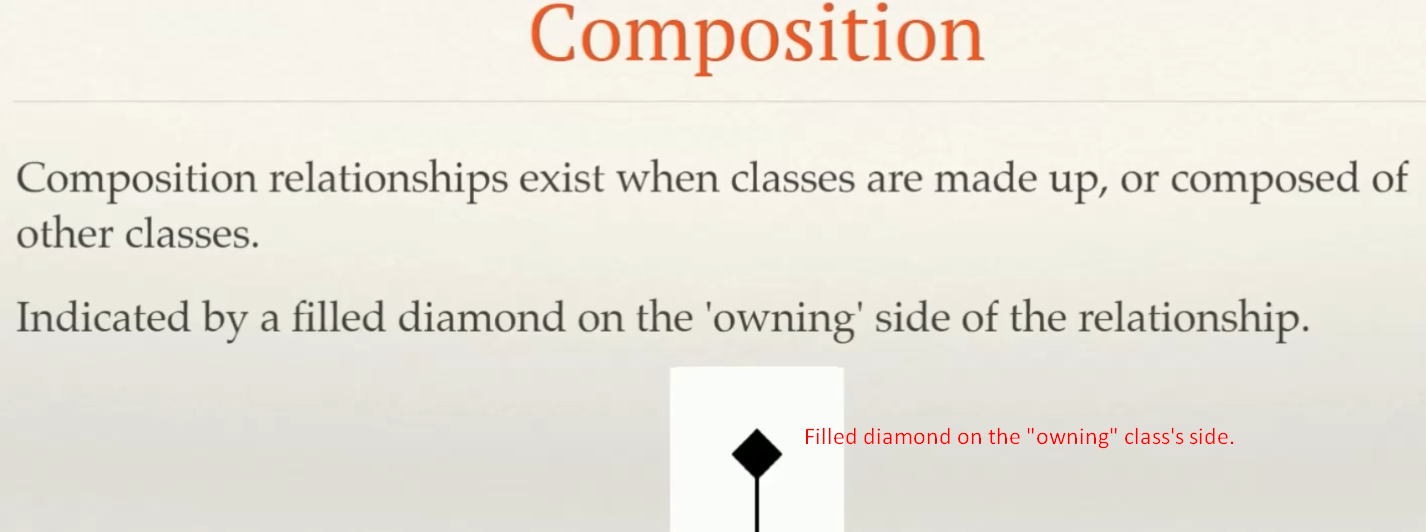
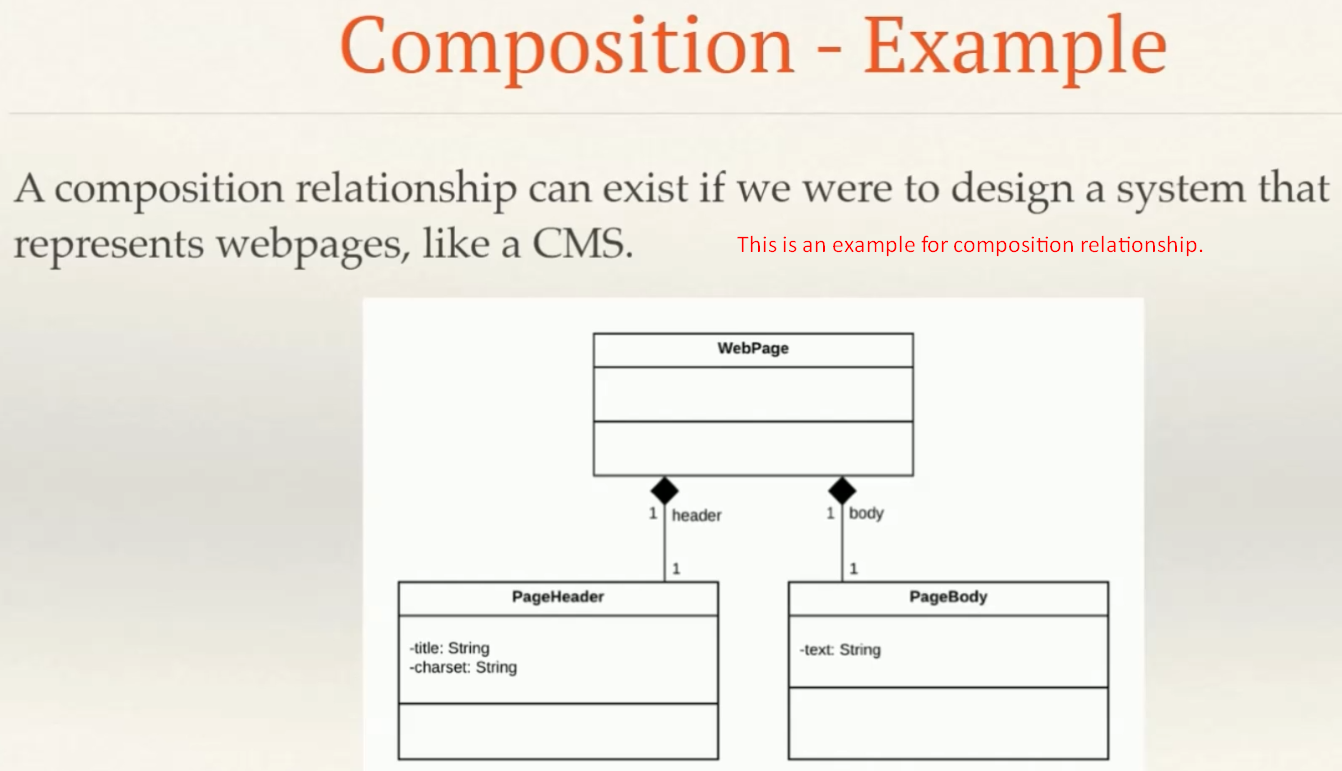
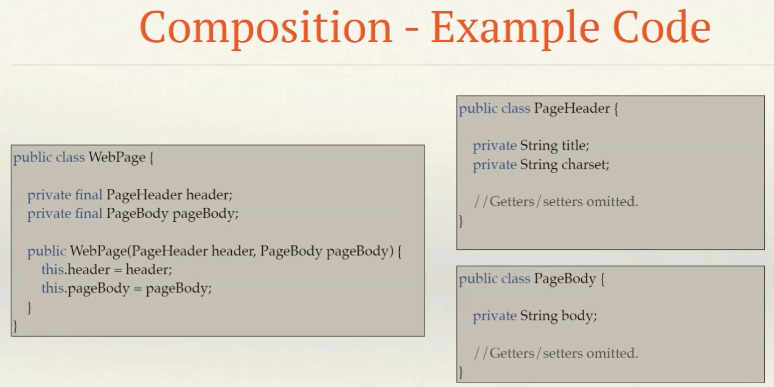
1. 

Association

1. Association relationship differs from Dependency Relationship in that a class doesn’t simply make use of another class (as in case of dependencies) - instead one class is explicitly coupled to another class because those classes communicate with each other.
2. Aggregation and composition relationships are specialized forms of association relationships.
3. 
4. 
5. Lastly we can indicate the role names on associations.
6. 
7.   
   See Doctor has property patients, it doesn’t mean that Doctor consists of Patients. So this is not aggregation or composition but just association.
8. 
9. 
10. 
11. In Code we represent the above relationship as follows:  
    **NOTE**:
    1. It may look similar to the doctor/patient example that we looked at earlier, but remember that doctors & patients are simply **associated** with one another.  **Jatin**: Doctor Class had the property List<Patient> but it doesn’t mean that those patients are part of a doctor (a doctor is consisting of patients).  
       A team on the other hand is consisting of a bunch of players. So, this is aggregation relationship.
    2. Keep in mind that you have to keep both sides of the relationship in **sync**.
12. 

Composition Relationship

1. **Composition Relationship**:  
   
   1. This relationship exists when classes are composed of other classes - in other words, objects of the “**owning**” class can’t exist without the child objects, and the child can’t exist without the **owning** object.  
      If the ‘owning’ object ceases to exists, so do the other objects.
   2. Composition and aggregation is a “**has a**” relationship.
   3. Composition is **represented** by filled diamond on the “**owning**” class’s side.
   4. 
   5. In code the classes can look like this.  
      
      1. Remember that our UML represents a conceptual view of the constraints that we apply to classes.
      2. In this example, we use final variables passed to a constructor to ensure that instances of the WebPage class can’t be created without a PageBody and PageHeader.
      3. However, if the code is written like this, it will still be possible to instantiate objects of the PageHeader and PageBody without necessarily having a parent WebPage object.
      4. Conceptually, this doesn’t make sense - but it’s up to you to decide how strictly you want to enforce design constraints when you write your code.  
         